uc3m | Universidad Carlos III de Madrid

Grado en Ingeniería Informática (2016-2017)

Trabajo Fin de Grado

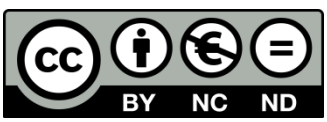
“Diseño e implementación de comportamientos inteligentes en StarCraft: Brood War”

Daniel Alejandro Rodríguez López

Tutora/es

Nerea Luis Mingueza

13-07-2017



Resumen

Los videojuegos se han convertido en los últimos años en el producto cultural que más beneficios aporta, por encima de la música y el cine. Esto es gracias a su constante crecimiento y multitud de géneros existentes. Uno de estos géneros, a su vez de los más antiguos son los juegos de estrategia, los cuales se pueden dividir en "Estrategia por turnos", por ejemplo el ajedrez, donde los jugadores tienen un turno, normalmente limitado a cierta cantidad de tiempo, para llevar a cabo sus movimientos o acciones; y "Estrategia en tiempo real", donde los jugadores juegan simultáneamente la partida teniendo que tomar multitud de diversas decisiones en un espacio de tiempo reducido. Los avances en la computación han hecho que esta segunda categoría tenga especial interés para el desarrollo de jugadores automáticos que usen Inteligencia Artificial, esto se debe a la necesidad de encontrar las mejores soluciones posibles en una cantidad de tiempo muy reducida, ya que la partida no puede detenerse por culpa de estos cálculos. Además, estos jugadores automáticos deben ser capaces de adaptarse a los cambios que ocurren a lo largo de la partida.

Este trabajo se centra en el videojuego "*StarCraft: Brood War*" y en el desarrollo de un agente con el objetivo inicial de ser capaz de derrotar a la Inteligencia Artificial del propio juego. Para el desarrollo se ha utilizado el lenguaje Java, haciendo uso de la API JNI-BWAPI y *ChaosLauncher* para poder conectarse con el juego. La principal técnica utilizada para el desarrollo del agente han sido árboles de comportamiento, apoyándose en mapas de influencia.

Abstract

In recent years videogames have become more profitable cultural products than music and movies together thanks to its constant change and multiple genres. One of these genres is Strategy games, which can be divided into two groups. First one is called Turn Based Strategy. Chess is for instance one of the games that belong to this group. Some features of this group include that players play in turns or limited time per turn. On the other hand, the other group is called Real Time Strategy, where all players play at the same time making multiple decisions in a short time. The advances in computing have made Real Time Strategy very attractive for the development of automatic players using Artificial Intelligence. This is interesting due to the need of finding the set of best suitable actions in a limit amount of time. Notice that the game is continuously evolving and never stops. Thus, the sooner the decisions are taken, the better. Besides, this *bots* must be able to adapt themselves to the changes happening in the game.

This work is focused in the videogame "*StarCraft: Brood War*" and the developing of an agent which aim is to be able to defeat the AI's game. The bot has been developed in Java using the JNI-BWAPI API and *ChaosLauncher* for game connection. The main technique used has been behavior trees, supported with influence maps.

Índice general

Índice de ilustraciones

Índice de tablas

Glosario de términos

- **Agente**: Bot que simula el comportamiento humano haciendo uso de técnicas de Inteligencia Artificial.
- **API**: **A**pplication **P**rogramming **I**nterface. Interfaz de programación de aplicación.
- **Bot:** Abreviatura de robot. Programa que realiza acciones de forma automática.
- **BWAPI**: **B**rood **W**ar **API**. *Framework* escrito en C++ que sirve para programar agentes para el videojuego *StarCraft: Brood War*.
- **ChaosLauncher**: Herramienta que permite utilizar conectar aplicaciones de terceros con el videojuego *StarCraft: Brood War*.
- ***Choke point***: Término utilizado para indicar que una zona del mapa actúa como cuello de botella. Es la unión de dos regiones y suelen ser puntos estratégicamente importantes.
- **Expandirse**: Acción de construir una base y recolectar recursos de otra zona diferente de la inicial.
- **Frame**: Imagen que se muestra en un instante. Es el equivalente a los fotogramas de las películas. Lo común es tener 30 ó 60 *frames per second* ó imágenes por segundo.
- **Inyección de código**: Técnica que consiste en enviar datos a una aplicación aprovechándose de vulnerabilidades de la misma.
- **JNI-BWAPI**: **J**ava **N**ative **I**nterface BWAPI. *Wrapper* de BWAPI que permite programar el agente en Java.
- **Macrogestión**: Abreviado como "Macro". Referido a toda la gestión relacionada con recursos, construcción y mejora de unidades, es decir, economía.
- **Microgestión**: Abreviado como "Micro". Referido a toda la gestión relacionada con el control de unidades: movimiento, ataques o uso de habilidades.
- **Niebla de Guerra:** Motivo por el que cierta región del mapa está oculta a ojos del jugador, por lo tanto, no sabe que ocurre en esa zona.
- **NPC**: ***N**on **P**layable **C**haracter*. En un videojuego son los personajes que no puede controlar el jugador.
- **Protoss**: Raza jugable en el videojuego *StarCraft*.
- **RTS**: Juegos de estrategia en tiempo real, del inglés *Real Time Strategy*. Género de videojuegos.

- ***Rush:*** Estrategia que consiste en entrenar rápidamente al comienzo de la partida un grupo de unidades con el que atacar e intentar vencer.
- **Scripting**: Consiste en la programación de eventos o comportamientos específicos.
- **StarCraft: Brood War**: Continuación del videojuego *StarCraft*, desarrollado por *Blizzard Entertainment* y puesto a la venta en Noviembre de 1998.
- **StarCraft**: Videojuego de estrategia en tiempo real desarrollado por *Blizzard Entertainment* y puesto a la venta en Marzo de 1998.
- **Terran:** Raza jugable en el videojuego *StarCraft*. Similar a los humanos.
- **Vespeno**: Recurso en el videojuego *StarCraft*. Se encuentra en géiseres que se encuentra a lo largo del mapa. Para poder extraerlo es necesario construir una refinería en el géiser.
- **Zerg**: Raza jugable en el videojuego *StarCraft*. Similares a insectos.

Capítulo 1: Introducción

En este capítulo se introduce el problema a tratar, la motivación detrás del mismo, los objetivos a alcanzar y la estructura del documento.

1.1. Descripción del problema

Los videojuegos se han convertido en uno de los pilares actuales de la industria del entretenimiento, siendo la principal fuente de ingresos para el sector del ocio [AEV, 2017]. Dentro de la multitud de géneros existentes podemos encontrar la estrategia en tiempo real (del inglés *Real Time Strategy*, abreviado como RTS), estos juegos se caracterizan por **enfrentar a los jugadores entre sí,** en un mapa cerrado, donde ambos jugadores realizan sus acciones **de forma simultánea**, es decir, no existen turnos y deben competir para cumplir cierto objetivo, como por ejemplo: destruir la base enemiga, controlar cierta zona del mapa durante un tiempo determinado o recolectar cierta cantidad de recursos. Otro aspecto a destacar dentro de este género es la complejidad del mismo, obligando al jugar a realizar y gestionar multitud de acciones y situaciones distintas a lo largo de la partida, obligando al jugador a adaptarse dependiendo del momento en el que se encuentre la misma. Si a esto se le añade la representación realizada dentro del propio juego se convierten en perfectos entornos para el desarrollo y prueba de técnicas de Inteligencia Artificial (IA), tales como búsqueda de caminos, optimización o desarrollo de comportamientos.

Entre los RTS existentes podemos encontrar **StarCraft** [Blizzard, 1998a], lanzado en 1998 y vigente a día de hoy. Se trata de uno de los juegos de estrategia más populares y con mayor número de jugadores de la historia [StarCraft AI, 2015]. Las características presentes en *StarCraft* que lo han hecho famoso tanto en el entretenimiento, competitivo como investigación son tres: mapas extensos con multitud de elementos, manejo de una gran cantidad de unidades y tres razas completamente diferenciadas. El aspecto más importante son las tres razas, donde cada una posee características únicas, lo que hace que la forma de jugar y enfrentarse a las mismas sea completamente distinta. Esto obliga a que dependiendo del mapa, la raza que controlemos y nos enfrentemos deba adoptar una estrategia u otra.

Para ofrecer un verdadero desafío al jugador, los *bots* o jugadores automáticos deberían ser capaces de adaptarse a la situación actual de la partida, lo cual implica contemplar una gran cantidad de parámetros, que pueden ser propios, del enemigo o del entorno. Sin embargo, la realidad es distinta, donde en vez de realizar agentes que se adapten a las situaciones, se pre-programan una serie fija de comportamientos [Blizzard, 1998b], los cuáles se eligen aleatoriamente al comienzo de cada partida, sin importar cómo juegue el humano.

La solución posible a este problema se puede encontrar realizando una aproximación mediante técnicas automáticas de Inteligencia Artificial como Redes de Neuronas, Clusterización o Algoritmos genéticos, o mediante otras más artesanales como puede ser Árboles de Comportamiento o Decisión o Máquinas de Estado.

1.2. Motivación

La Inteligencia Artificial presente en los videojuegos cada vez es más completa, pero no porque las técnicas utilizadas sean nuevas o muy complejas, si no principalmente, por la combinación de técnicas o uso de técnicas no utilizadas hasta ahora sobre un dominio concreto.

Este avance se debe principalmente a la mejora de capacidad y velocidad de cómputo en los ordenadores, permitiéndoles manejar una cantidad de datos que hace 15-20 años era inimaginable. Pero también se debe a la exigencia de los jugadores y a la "profesionalización" de los videojuegos debido a la popularización y normalización de los deportes electrónicos teniendo torneos oficiales donde los premios llegan a ser de varios millón de dólares [eSports prizes], por lo que la necesidad de mejorar continuamente está más presente que nunca y para ello se necesitan inteligencias artificiales más exigentes y desafiantes. También existen torneos donde se enfrentan varios *bots* entre ellos (SSCAIT [SSCAIT, 2017] o AIIDE [AIIDE, 2017]), donde continuamente se están enviando *bots* desarrollados, de forma que se compita por ver quien define los mejores comportamientos.

Inicialmente, para hacer a los jugadores controlados por la máquina más desafiantes, se les daba un conocimiento absoluto de los datos, es decir, el enemigo sabía todo lo que hacías, incluso aunque no te viese. Esto se puede ver clarísimamente en numerosos juegos de antes del 2000 como *Civilization* o *StarCraft* y años posteriores e incluso en juegos más actuales [The All Seeing AI].

Si bien esto puede suponer un desafío, ya que se está en desventaja frente al enemigo, no representa la situación real a la que se encontraría un jugador cuando juega contra otro ser humano. Por ello el desarrollo de jugadores automáticos que jueguen de forma limpia y sean desafiantes es todo un reto, ya que se deben establecer técnicas para obtener toda esa información desconocida de forma justa, pero al mismo tiempo, debe adaptarse e intentar derrotar al jugador y por encima de todo: debe parecer humano.

1.3. Objetivos del trabajo

El objetivo principal del trabajo es la realización de un **Agente** inteligente aplicando técnicas de Inteligencia Artificial. El agente deberá ser capaz de vencer a

la IA del juego y que se pueda adaptar a situaciones dependiendo del mapa y enemigo al que se enfrente. Esto implica contemplar e incluir en dicho agente:

- **Análisis del entorno**: Para poder tomar decisiones tales como: Donde construir, recolectar recursos, enviar tropas al ataque ó expandirse.
- **Toma de decisiones**: Ya que como se ha explicado en la Introducción, cada raza funciona de forma completamente distinta a las demás.
- **Gestión de ataque y defensa**: Es el factor más importante, sin una buena técnica de ataque y defensa nunca se podrán ganar partidas.

1.4. Estructura del documento

En este apartado se realiza una breve descripción del contenido de cada sección del presente documento:

- Capítulo 1: Introducción. En este capítulo se realiza una introducción al documento, describiendo brevemente el problema y planteando los objetivos del trabajo.
- Capítulo 2: Estado del Arte. En este capítulo se describe la situación actual del Estado del Arte, sobre todos los aspectos relacionados con el trabajo.
- Capítulo 3: Descripción del sistema. En este capítulo se realiza una descripción detallada de todos los componentes presentes en el trabajo, así como una visión general de la arquitectura del sistema.
- Capítulo 4: Experimentación. En este capítulo se muestra y analiza la experimentación realizada para verificar el correcto funcionamiento del trabajo y su evolución a lo largo de las distintas versiones desarrolladas.
- Capítulo 5: Gestión del proyecto. En este capítulo se explica, detalla y justifica la metodología de desarrollo elegida para el proyecto. Esta sección incluye además un presupuesto del mismo, el marco regulador existente actualmente para el desarrollo software y el impacto socio-económico del mismo.
- Capítulo 6: Conclusiones y trabajos futuros. En este capítulo se exponen las conclusiones extraídas del proyecto, conclusiones respecto a los objetivos planteados en este mismo capítulo y trabajos futuros que se podrían realizar sobre este trabajo.
- Chapter 7: Design and development of behaviors in StarCraft: Brood War Este capítulo contiene la introducción, objetivos, resultados y conclusiones en inglés.
- Capítulo 8: Anexos. Este capítulo contiene el “Manual de instalación”, el cual indica como instalar este trabajo, el “Manual de usuario”, el cual indica cómo utilizarlo y todos los datos extraídos de las pruebas realizadas.

Capítulo 2: Estado del Arte

Este capítulo contiene una introducción a los videojuegos de estrategia, centrándose específicamente en *StarCraft*, del cual se detallará su funcionamiento.

Posteriormente se realiza un repaso por la historia de la Inteligencia Artificial (IA), explicando las técnicas fundamentales para luego centrar la atención en aplicaciones de la misma en los videojuegos.

A continuación se habla sobre trabajos relacionados con *StarCraft* e IA, donde se podrán ver las distintas aproximaciones y esfuerzos realizados.

Finalmente se describe la situación actual del desarrollo de agentes para *StarCraft*, comentando tanto la existencia de herramientas para su desarrollo como competiciones actuales.

2.1. Juegos de Estrategia y StarCraft

Si pensamos en juegos de estrategia tradicionales podemos pensar en ejemplos como el Ajedrez, Damas o Go, juegos de mesa que existen desde hace siglos, pero también podemos viajar a una época más actual y encontrarnos juegos de miniaturas como *Warhammer* [Games Workshop, 1983], de cartas como *Yu-Gi-Oh!* [Konami, 1999] o *Magic: The Gathering* [Wizards of the Coast, 1993] o de mesa como *Catán* [Kosmos, 1995] o *¡Pingüinos!* [999 Games, 2003] Pese a que sus reglas de juegos son muy distintas y no son aplicables entre sí, tienen en común que todos y cada uno son juegos de estrategia con una serie de características comunes que podemos extraer:

1. Se enfrentan dos o más jugadores.
2. El objetivo es derrotar al enemigo, ya sea eliminando su ejército (Ajedrez, Damas o Warhammer), consiguiendo el control de la mayor parte del tablero de juego (Go o *¡Pingüinos!*), eliminando sus "Puntos de vida" (*Yu-Gi-Oh!* o *Magic: The Gathering*) o cumpliendo una serie de objetivos (*Catán*).
3. Todos ellos transcurren en un mapa o tablero de juego, caracterizado por poseer zonas o regiones específicas más o menos definidas, yendo desde simples casillas (Ajedrez, Damas o Go) hasta regiones más elaboradas como puede ser un escenario real (*Warhammer*).
4. A lo largo del juego los jugadores realizan ciertas acciones que modifican el estado del juego.

Estas tres características presentes en juegos tradicionales se pueden aplicar perfectamente a videojuegos, siendo los más populares los de carácter bélico, podría decirse, equivalentes al Ajedrez. Entre estos tipos de juegos se encuentran dos categorías principales: juegos de **Estrategia por Turnos (TBS)** y Juegos de

Estrategia en Tiempo Real (RTS). Dentro de ambos géneros, se diferencia entre **juegos de gestión y juegos de guerra**. Los juegos de gestión permiten al jugador controlar todos los aspectos de la partida: recolección de recursos, creación de edificios, creación y mejora de tropas y control sobre las unidades militares; por otro lado, los juegos de guerra carecen de los aspectos de recolección de recursos y creación de edificios, pasando directamente al control de unidades militares, y permitiendo normalmente, la mejora y/o personalización de las unidades.

A su vez estos, juegos de gestión y guerra, se agrupar en dos grandes grupos dependiendo de cómo se obtenga la información: **juegos de información perfecta**, donde ambos jugadores conocen toda la información del oponente, por ejemplo Ajedrez, y **Juegos de información imperfecta**, donde el jugador sólo conoce su propia información y la información del enemigo se encuentra oculta en la llamada **Niebla de Guerra** (del inglés *Fog of war*), por lo que el jugador debe explorar para ir obteniendo parcialmente esa información.

Los TBS son los más similares al Ajedrez. Dos o más jugadores juegan en un mapa donde, en cada uno de sus turnos, a veces limitados por el tiempo, pueden realizar una serie determinada de acciones y una vez finalizadas pasan el turno al siguiente jugador. El ejemplo actual referente de los juegos de gestión TBS es la saga ***Sid Meier's Civilization*** [Microprose, 1991], abreviado como Civilization (Ilustración 1).

Ilustración 1 Captura de Civilization VI (Fuente

En él, los jugadores se ponen al frente de una civilización, por ejemplo azteca, japonesa o griega entre muchas otras, y deben alcanzar uno de los tres objetivos posibles para ganar: militar, eliminando al resto de enemigos, ciencia, ganando la carrera espacial, o cultural, construyendo una serie de edificios importantes.

Por otro lado, como juego de guerra referente podemos tomar la saga ***Fire Emblem*** [Intelligent Systems, 1990] (Ilustración 2).

Ilustración 2 Captura de Fire Emblem: Awakening (Fuente

En él, el jugador dirige un grupo limitado de unidades, las cuales pueden morir permanentemente para el resto del juego. A lo largo del juego, el jugador debe vencer al enemigo cumpliendo el objetivo marcado, normalmente eliminar todas las unidades enemigas o al comandante.

Los RTS son también similares al Ajedrez, pero con la característica de que no existen los turnos. Ambos jugadores realizan sus acciones de forma simultánea a lo largo de la partida. Estos juegos tuvieron su época dorada a finales de la década de los 90, con juegos como ***Warcraft*** [Blizzard Entertainment, 1994], ***Tzar*** [Haemimont Games, 1997] o ***Age of Empires*** [Ensemble Studios, 1997], todos juegos de gestión, que sentaron las bases del género estableciendo una serie de características comunes desde entonces:

- La recolección y gestión de los recursos es fundamental, ya que son la base para el resto de acciones en el juego.
- Mapas grandes, con numerosos elementos: bosques, mar, islas, elevaciones…
- Posibilidad de elegir entre varias razas o civilizaciones, todas ellas con unidades distintas, ya sea orcos o humanos en *Warcraft*, civilizaciones europeas, árabe o asiática en *Tzar* o griega, fenicia, egipcia o persa entre muchas otras en *Age of Empires*.
- Control de un gran número de unidades, se podía tener fácilmente al mismo tiempo más de 20 unidades distintas.

Todos estos juegos sentaron las bases para que en 1998 surgiese el juego referencia, ***StarCraft***. Dado que el trabajo trata sobre el mismo, se habla detalladamente más adelante en la subsección 2.1.1. *StarCraft*.

Por otro lado, los ejemplos más destacables de RTS de guerra son relativamente nuevos, y poseen una peculiaridad a diferencia de los de tipo gestión. El transcurso de las peleas ocurren en tiempo real, mientras que la gestión de unidades y movimientos entre territorios son por turnos. Destacan sobre todo la saga ***Total War*** [The Creative Assembly, 2000] (Ilustración 3) o el más reciente ***Dawn of War III*** [Relic Entertainment, 2017].

Ilustración 3 Captura de Total War: Warhammer (Fuente

En ambos juegos el movimiento de tropas a lo largo del mapa y la gestión de las mismas se realiza por turnos, mientras que los enfrentamientos contra los enemigos se realiza en combates en tiempo real. Particularmente, además se puede detener el tiempo para realizar acciones como órdenes de ataque o movimiento, se puede seleccionar entre distintas civilizaciones o razas. La gestión de los recursos pasa a un plano secundario. La diferencia fundamental entre ambos juegos es que en la saga *Total War* el número de unidades presentes en los combates pueden ser decenas, mientras que en *Dawn of War III* el número de unidades a manejar suele ser normalmente del orden de cinco unidades distintas.

2.1.1. StarCraft

Desarrollado por **Blizzard Entertainment** y puesto a la venta en 1998, el videojuego de estrategia en tiempo real (RTS) ***StarCraft*** y posteriormente ***StarCraft: Brood War***, han vendido conjuntamente a lo largo de su historia más 10 millones de unidades a lo largo de todo el mundo [IGN, 2008], convirtiéndose en un referente del género desde entonces.

Al igual que en la gran mayoría de juegos de estrategia, el jugador debe recolectar recursos, entrenar unidades y derrotar al enemigo. Concretamente, en *StarCraft* se

pueden controlar tres razas completamente diferenciadas: **Terran** (Ilustración 4)**, Protoss** (Ilustración 5) **y Zerg** (Ilustración 6)**.** Cada raza cuenta con unidades, edificios, habilidades y tecnología única, lo que hace que tanto jugar con ella, como contra ella sea completamente distinto.

Ilustración 4 Terran
(Fuente)

Ilustración 5 Protoss
(Fuente)

Ilustración 6 Zerg
(Fuente)

Como se ha comentado previamente, existían otros juegos como *Warcraft* o *Age of Empires* que tenían muchas características en común con *StarCraft*, pero fue en éste cuando el control de un número elevado de unidades se facilitó, dando lugar a peleas a gran escala o permitiendo al jugador atacar varios frentes al mismo tiempo, aumentando así las posibilidades que ofrecían hasta ese momento todos los RTS aumentando la relevancia en las estrategias en el ataque y defensa.

Dentro del juego las unidades de cada raza se pueden dividir en dos tipos:

- Trabajadoras: Son las unidades que se encargan de la recolección de recursos y de construir edificios. Se entrenan en el edificio base. Cada raza tiene su propia unidad: VCE (Vehículo de Construcción Espacial) para los Terran, Sondas para los Protoss y Zánganos para los Zerg.
- Militares: Son el resto de unidades.

Para poder entrenar (crear) unidades, construir edificios y realizar mejoras, se tienen tres tipos de recursos, que se pueden visualizar en la Ilustración 7:

Ilustración 7 Recursos y Suministros (Fuente

- Mineral: Necesario para cualquier construcción, unidad o mejora. Se recolectan con constructores, los cuales deben entregarlo en su base.
- Gas Vespeno: Se recolecta a partir de géiseres de vespeno. Se suele utilizar para unidades o edificios avanzados y mejoras. Para poder recolectarlo, antes es necesario construir un edificio especial dedicado a la recolección de gas vespeno, el cual depende de la raza.
- Suministros: Cada unidad consume cierta cantidad de suministros. Existe un límite de los mismos que se puede aumentar hasta 200 construyendo el edificio o entrenando la unidad adecuada según la raza. Los Terran aumentan sus suministros construyendo "Depósitos de suministros", los Protoss construyendo "Pilones" y los Zerg entrenando "Superamos".

Todas las razas tienen su árbol tecnológico, es decir edificios, dividido en dos grandes grupos: edificios básicos y edificios avanzados. La diferencia principal es que los Edificios básicos cuestan solo mineral, y los Edificios avanzados cuestan también vespeno.

2.1.1.1. Terran

Los Terran son el equivalente a los humanos dentro del juego. Sus unidades suelen ser humanos, vehículos o ro*bots*. Algunos ejemplos de unidades se pueden ver en la Ilustración 8 y su árbol tecnológico en la Ilustración 9.

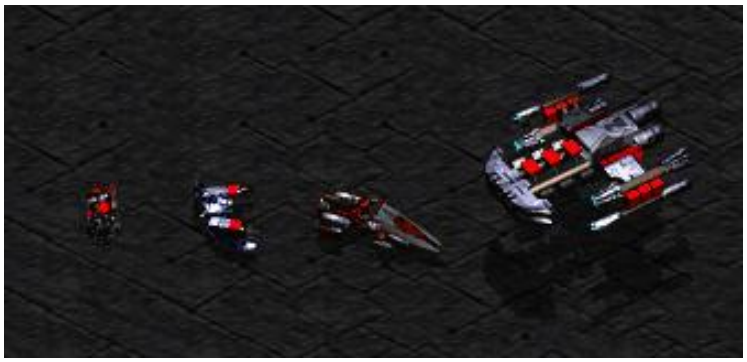


Ilustración 8 Ejemplo de unidades Terran

Poseen las siguientes características únicas:

- Sus edificios pueden construirse en cualquier zona del mapa, siempre y cuando el edificio quepa, todo el terreno esté a la misma altura, no haya huecos y no exista biomateria.
- Sus edificios pueden repararse.
- Tiende a bloquear el camino construyendo edificios en él, de forma que el enemigo no pueda pasar fácilmente sin antes destruirlos.
- Todos los edificios que producen unidades, además de la *Engineering Bay* pueden elevarse y desplazarse una vez construidos.
- Si el edificio está muy dañado, comienza arder y pierde vida lentamente.
- Casi todas las unidades pueden sanarse, ya sea reparándolas (por ser vehículos) o con médicos.
- Puede construir *Bunkers* que permiten a las unidades protegerse mientras atacan.

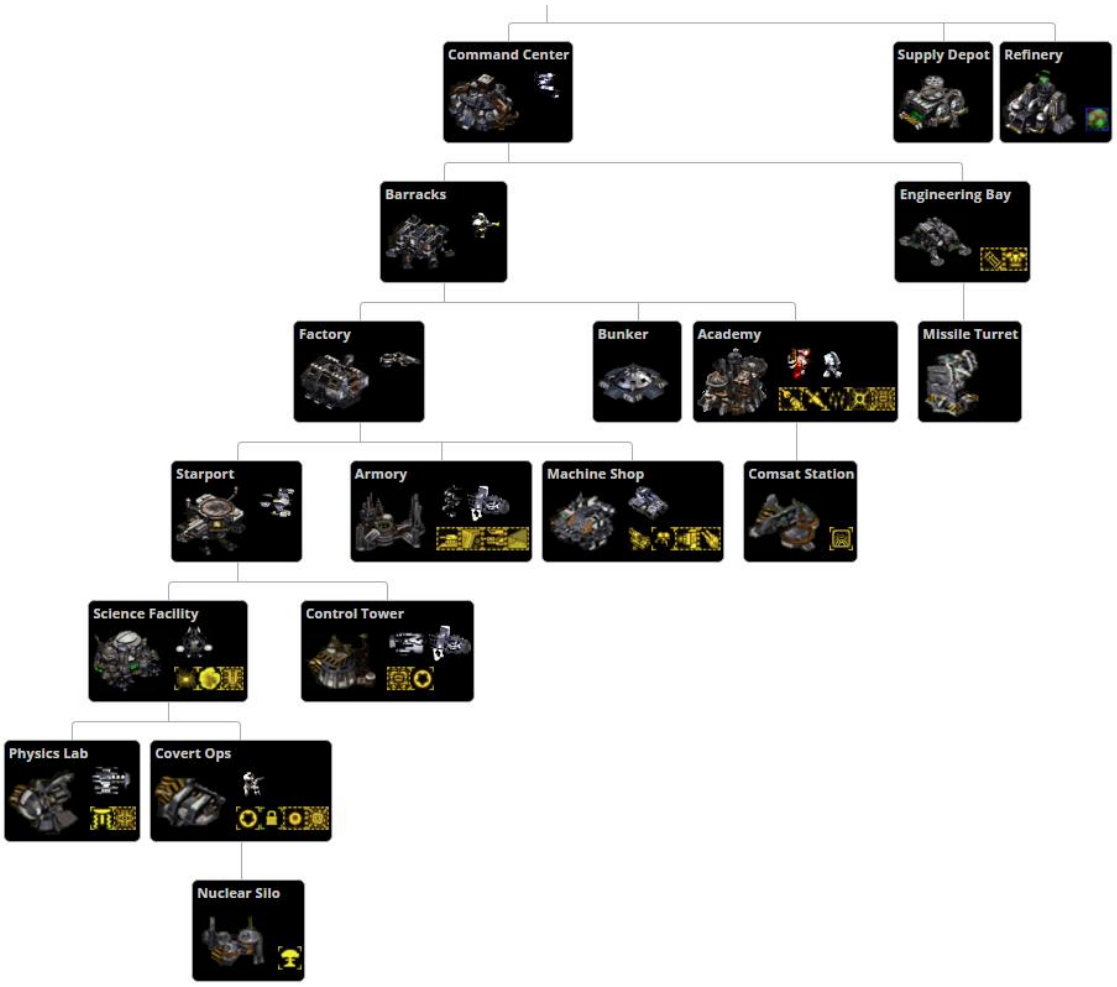


Ilustración 9 Árbol tecnológico Terran (Fuente

En cuanto a las unidades más comunes se suelen entrenar:

- Marine: Unidad terrestre de ataque a distancia. Suelen ser la unidad principal combinada con médicos.
- Médicos: Unidad terrestre que puede curar a otras unidades de tipo orgánico (es decir, no vehículos), pero no puede atacar.
- Murciélago de fuego: Unidad terrestre de corto alcance, pero con daño en área, es decir, puede dañar a varias unidades a la vez. Muy efectiva contra *Zergs*.
- Buitre: Vehículo terrestre, barato con gran movilidad y alcance, si se juega correctamente puede causar muchísimo daño al enemigo.
- Tanque de asedio: Vehículo terrestre, principalmente defensivo, produce gran daño en comparación con otros similares. Su principal característica es que tiene dos modos: tanque y asedio, en modo Tanque el vehículo puede moverse, pero tiene un corto alcance, por otro lado, en modo Asedio está inmovilizado, pero su alcance aumenta considerablemente y realiza daño en área, siendo muy efectivo tanto para defender como para atacar y mantener al enemigo confinado.
- Nave de evacuación: Vehículo aéreo que permite cargar y descargar unidades. Se suele utilizar para transportar unidades de forma rápida, pero

sobre todo, para atacar la retaguardia del enemigo, causar daños y escapar antes de que el enemigo llegue con sus unidades a defender. Suele utilizarse principalmente para destruir a los trabajadores del enemigo y así cortar su línea de suministros.

En cuanto al combate, las tácticas más conocidas y usadas son:

- *"Sim City"*: Consiste en construir edificios para obstaculizar el camino del enemigo, muy utilizada tanto contra Protoss como contra Zerg, ya que sus unidades iniciales son cuerpo a cuerpo, mientras que los marines son a distancia.
- "Masa de marines y médicos": Consiste en entrenar un gran número de marines, apoyados por médicos, esto debe ir acompañado de la mejora "Paquete de estimulantes" la cual aumenta la velocidad de ataque de los marines, pero les hace daño. La combinación Marines + Médicos suele ser muy común dada su fuerza y aguante en las peleas, pudiendo vencer frente a numerosos enemigos en inferioridad numérica.
- "*Dropship*": Consiste en crear un grupo de unidades, pequeño o grande y apoyándose de naves de evacuación atacar continuamente ("*Harassing*") a los trabajadores del enemigo o distintas zonas del mapa, obligando al enemigo o bien a dividirse o bien a atacar un único punto. Sin embargo, cuando el enemigo se acerca, se cargan las unidades de nuevo en la nave de evacuación y se marchan, provocando así daños considerables sin perder unidades.

2.1.1.2. Zerg

Los *Zerg* son una raza alienígena, caracterizada por su similitud a los reptiles. Algunos ejemplos de unidades se pueden ver en la Ilustración 10 y su árbol tecnológico en la Ilustración 11.

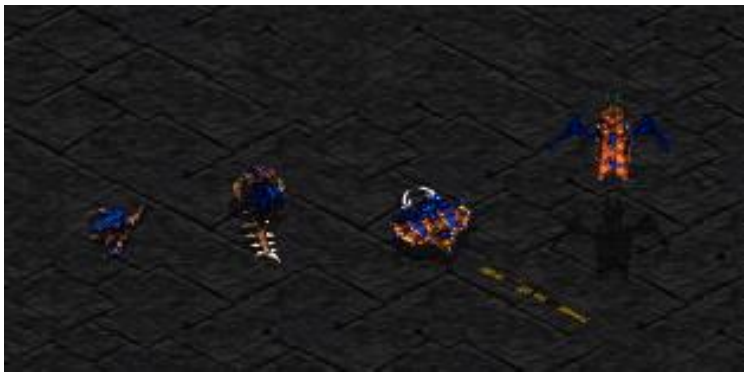


Ilustración 10 Ejemplos de unidades Zerg

Poseen las siguientes características únicas:

- Sus unidades son las más baratas del juego. También las más débiles en comparación con las otras razas.

- Sólo posee un único edificio de producción, la base, desde donde puede entrenar cualquier tipo de unidad una vez desbloqueada.
- Los zánganos (constructores) mueren al construir un nuevo edificio.
- Sólo se puede construir edificios encima de biomateria, la cual es generada por la base y por el edificio *Creep colony*.
- Todas las unidades, incluidos edificios, regeneran salud lentamente.
- Todas las unidades terrestres pueden enterrarse en el suelo.
- Algunas unidades pueden mutar (transformarse) en otras.
- Su base puede mejorarse: de *Hatchery* a *Lair* y de *Lair* a *Hive*.

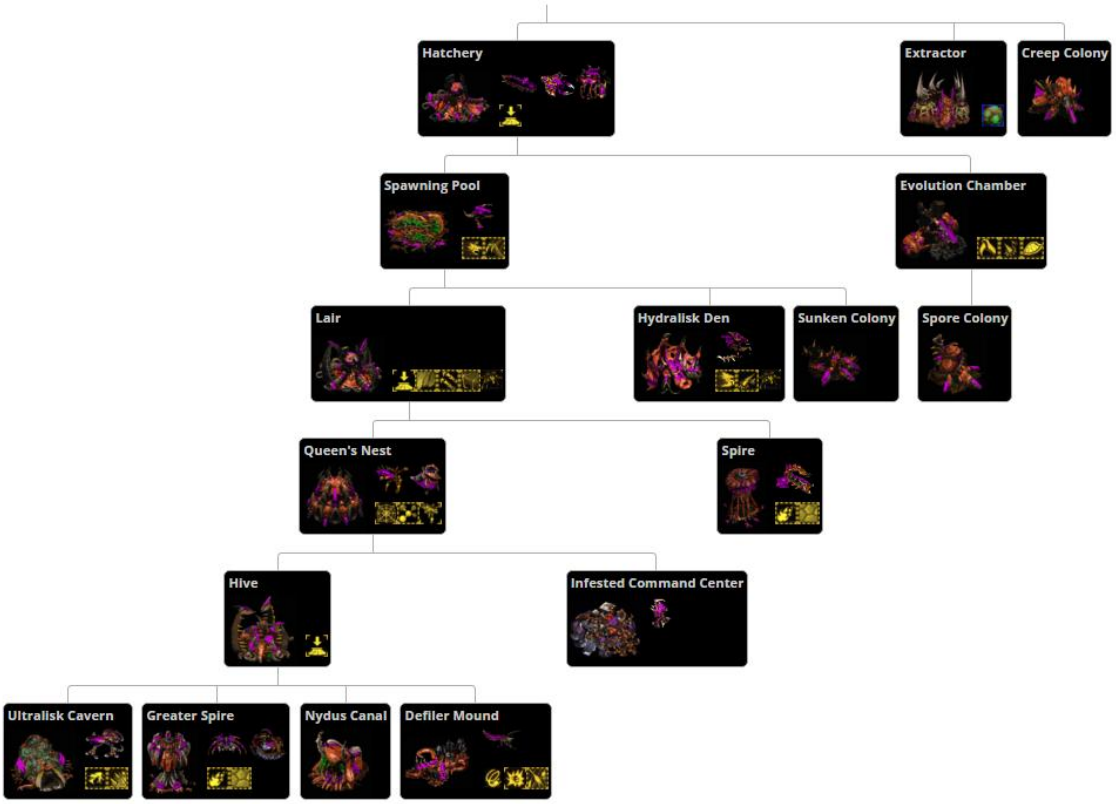


Ilustración 11 Árbol tecnológico *Zerg* (Fuente

En cuanto a las unidades más comunes se suelen entrenar:

- Zerlings: Unidad terrestre de ataque cuerpo a cuerpo. Es la unidad básica de los *Zerg*, se caracterizan por ser baratas, poseer poca vida y entrenarse de dos en dos. Además posee gran movilidad.
- Superamos: Unidad aérea que no puede atacar. Cada superamo aumenta en 8 el límite de suministros. Se mueven lentamente y son unidades detectoras, es decir, revelan unidades invisibles. Además pueden mejorarse para poder cargar/descargar unidades y moverse más rápido.
- Hidraliscos: Unidad terrestre de ataque a distancia. Su relación coste-daño-movilidad la convierte en una unidad muy común, sobre todo contra *Terran* y *Protoss*. Además, puede mutar en Merodeador.
- Merodeador: Unidad terrestre de ataque a distancia (sólo terrestre). Sólo puede atacar cuando está enterrada. Realiza un ataque en fila que daña a

todos los enemigos a su paso. Muy útil para defender contra ataques iniciales dado que al estar enterrada se necesita de detectores para ser revelada.

- Mutaliscos: Unidad aérea de ataque a distancia. Su ataque rebota entre los enemigos y posee gran movilidad por lo que suele utilizarse para atacar a los trabajadores del enemigo.
- Ultralisco: Unidad terrestre de ataque cuerpo a cuerpo. Es la unidad más poderosa de los *Zerg* y también de las más caras. Posee un ataque en área y es muy resistente, por lo que se suele entrenar al final de la partida.

En cuanto al combate, las tácticas más conocidas y usadas son:

- "Expansión temprana": Dado que los *zergs* sólo pueden producir unidades en el *Hatchery*/*Lair*/*Hive*, este edificio es ligeramente más barato que el de las otras razas, por lo que se suele construir rápidamente otro para aumentar la producción de unidades y la recolección de recursos. Es arriesgado debido a que implica ahorrar una gran cantidad de mineral, por lo que normalmente se dejaría de entrenar unidades militares en ese rato.
- "*Six pool*": Esta técnica es la más conocida y arriesgada, pero si sale bien suele dar la partida al jugador *Zerg*. Consiste en centrarse en entrenar tres pares de zerlings, dejando de lado la defensa y economía hasta ese momento. Una vez se tienen entrenados los seis zerlings, se ataca con ellos la base del enemigo. Si el enemigo no está preparado para los zerlings podrán destruir todos sus trabajadores y por tanto, destruir su economía y ganar la partida.
- "Micro de mutaliscos": Consiste en entrenar un gran número de mutaliscos, agruparlos y atacar realizando movimientos rápidos y cortos, de esta forma se maximiza la velocidad de ataque y dado que el ataque de los mutaliscos rebota, el daño total causado es considerable.

2.1.1.3. Protoss

Los *Protoss* son una raza alienígena, muy por encima tecnológicamente tanto de los *Terran* como de los *Zerg*. Algunos ejemplos de unidades se pueden ver en la Ilustración 12 y su árbol tecnológico en la Ilustración 13.

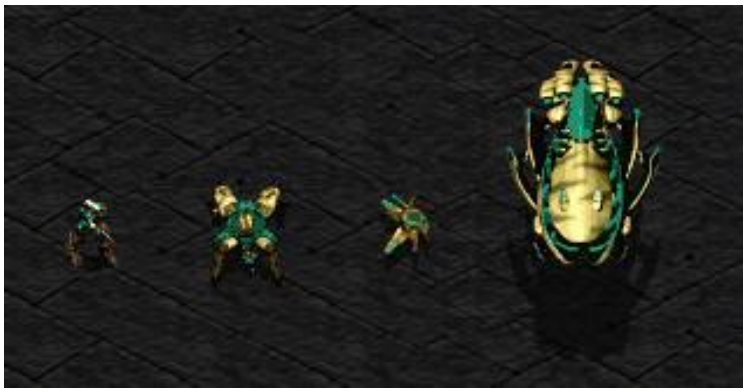


Ilustración 12 Ejemplos de unidades Protoss

Poseen las siguientes características únicas:

- Todas sus unidades, incluidos edificios, poseen un escudo que se regenera con el tiempo si es dañado.
- Sus unidades son las más caras en comparación con el resto de razas.
- La construcción de edificios no requiere que un trabajador (Sonda) esté ocupado construyéndolo, una vez se inicia la construcción, se finaliza sola, lo que deja a la sonda libre para realizar otras tareas.
- Para poder construir, los edificios deben situarse dentro del campo de energía. Este campo es generado por los *Pylons*. Si un edificio se queda sin campo de energía (por ejemplo, un pilón es destruido), deja de funcionar.
- Posee unidades que permanecen permanentemente en invisible: "Observador" y "Templario Tétrico".

En cuanto a las unidades más comunes se suelen entrenar:

- Fanáticos: Unidad terrestre de ataque cuerpo a cuerpo. Se suele entrenar al comienzo de la partida debido a su gran aguante tanto frente a marines como *zerlings*.
- Dragones: Unidad terrestre de ataque a distancia. Se combina con fanáticos para cubrir su debilidad frente vehículos aéreos o ataques a distancia.
- Templarios Tétricos: Unidad terrestre de ataque cuerpo a cuerpo e invisible. Se utiliza principalmente para destruir los trabajadores del enemigo. Son muy débiles, por lo que si son detectados suelen morir rápido.
- Arconte y Arconte Tétrico: Unidad terrestre de ataque a distancia. Unidad poderosa que se entrena "fusionando" dos altos templarios o dos templarios tétricos respectivamente. Su gran ventaja es el gran escudo que posee, por el contrario tienen muy poca vida.
- Transportes: Vehículo aéreo que no puede atacar. Para atacar entrena (cuesta mineral) unas pequeñas naves que atacan y pueden ser destruidas, lo que le permite atacar desde lejos. Se entrenan sobre todo a mediados-final de la partida debido a su gran coste y tiempo de entrenamiento.

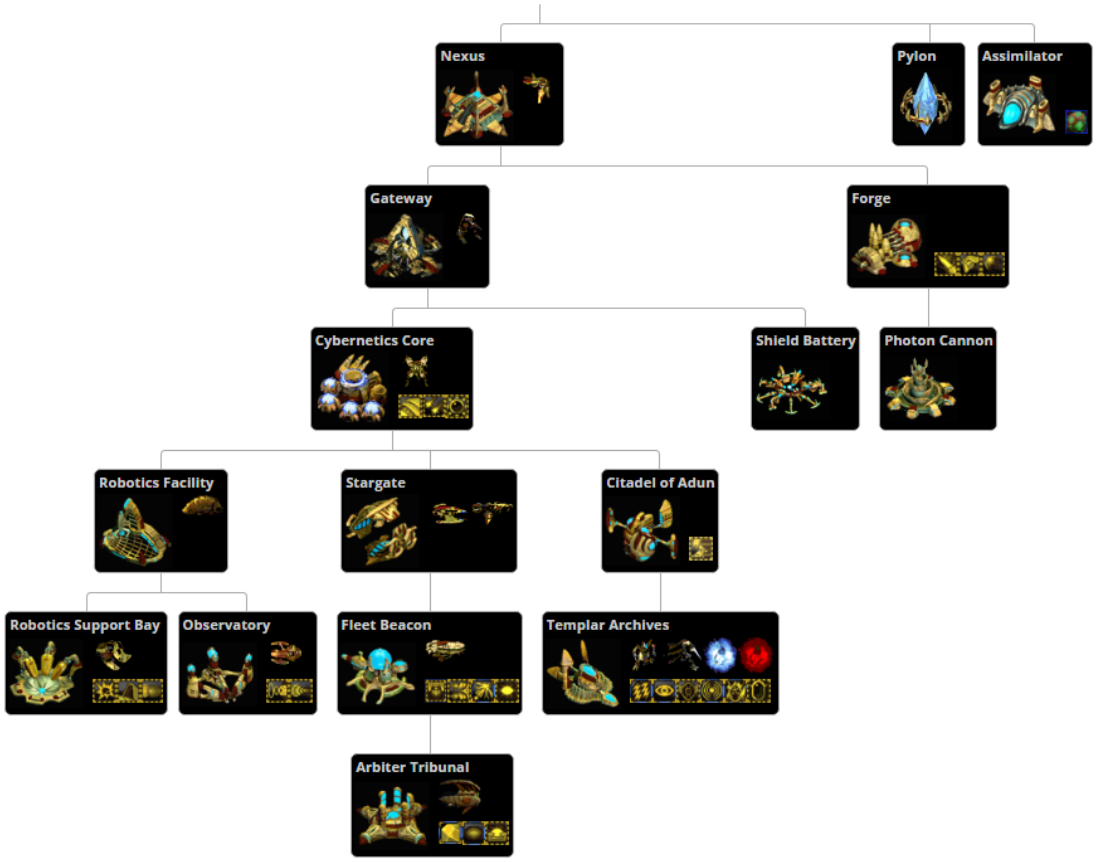


Ilustración 13 Árbol tecnológico Protoss (Fuente

En cuanto al combate, las tácticas más conocidas y usadas son:

- *"Sim City"*: Consiste en construir edificios para obstaculizar el camino del enemigo, muy utilizada debido a que se poseen edificios capaces de atacar a enemigos, tales como los "Cañones de fotones".
- "Fanáticos, Dragones y Arcontes": Dado que los fanáticos son unidades poderosas, pero sólo pueden atacar unidades terrestres, se combinan con dragones para compensar esa debilidad. Adicionalmente, si la partida se alarga, se complementa también con arcontes, que son unidades muy poderosas.
- *"Cannon rush"*: Táctica arriesgada que consiste en construir cañones de fotones en las líneas de mineral tanto de la base como de las expansiones naturales del enemigo, de esta forma se corta la obtención de mineral.

2.1.1.4. Mapas

Por último, los mapas también han jugado un papel importante en la consolidación de *StarCraft* sobre todo debido a su tamaño, teniendo mapas de hasta 256x256 píxeles, mientras que WarCraft II o Age of Empires (juegos del mismo periodo) tenían de hasta 128x128 píxeles.

Adicionalmente, en los mapas se pueden destacar tres características:

- Distintos niveles de altura, haciendo que un jugador en un terreno a mayor altura tenga visión de los niveles inferiores, pero no al revés.

- Cuellos de botella o "*Choke Points*": Zonas del mapa que sirven para separar y conectar distintas regiones, convirtiéndolos en puntos de defensa importantes.
- Expansiones: Zonas específicas del mapa con recursos (mineral y gas vespeno) que el jugador puede explotar de forma eficiente si construye una base en esa localización, repercutiendo positivamente en la economía del jugador. El proceso de creación de una expansión puede observarse en la Ilustración 14. Al tratarse de zonas específicas, serán puntos vulnerables del mapa y normalmente no se encontrarán tan bien protegidas por defecto como la localización inicial. La decisión de cuándo construir o no una expansión repercute bastante a lo largo de la partida. Si se hace muy pronto, debido al coste que implica, se dejan de fabricar unidades militares y por lo tanto la defensa será menor en la base inicial en cuanto a los primeros ataques ofensivos de la partida, pero si se hace muy tarde, es probable que el enemigo ya se haya expandido más de una vez.

Ilustración 14 Proceso de construcción de una expansión

Todas estas características y toma de decisiones y escenarios posibles, junto con el hecho de poseer tres razas completamente diferenciadas y perfectamente balanceadas, donde cada raza tiene unidades mejores y peores contra otras, lo convierte en todo un desafío, tanto para jugadores profesionales, como para el desarrollo efectivo de jugadores automáticos o *bots* .

2.2. Inteligencia Artificial

Este apartado se divide en tres partes. En primer lugar se describe brevemente la historia de la IA. Después se explican en detalle diversas técnicas de IA que además se han utilizado en juegos de estratega. Finalmente se relaciona cada una de esas técnicas con uno o varios ejemplos de su uso dentro de estos juegos, justificando su uso más común en cada género.

2.2.1. IA en la historia

Las primeras técnicas de IA se empezaron a utilizar y comenzaron a desarrollarse a partir de los años 50, cuando Alan Turing publicó su artículo "Computing *machinery and intelligence"* donde se plantea por primera vez el conocido "Test de Turing": ¿Puede una máquina llegar a imitar/hacerse pasar por un ser humano? [Stuart y Russell, 2004]

Como se explica en [Stuart y Russell, 2004], desde los años 50 hasta finales de los 60, la IA se comenzó a utilizar para, sobre todo, solucionar principalmente problemas sencillos de planificación y matemáticas. Durante estos años, John McCarthy, uno de los padres de la IA, desarrolló el lenguaje de programación **Lisp**, pensado inicialmente para programar módulos de IA. Sin embargo, la falta de recursos en los computadores era una limitación física que afectaba directamente a la dificultad de los problemas que se pretendían resolver. Durante estos años se crearon sistemas de planificación, demostradores matemáticos y las primeras **Redes de Neuronas Artificiales** (Adaline y Perceptrón), aunque habían sido descritas inicialmente en 1949 por McCulloch y Pitts.

Tras los prometedores resultados resolviendo problemas sencillos, la comunidad científica tenía grandes esperanzas en la IA. Sin embargo, su aplicación a problemas poco conocidos (como la traducción de documentos), intratables (dominios muy grandes) y la limitación de cómputo implicó que los resultados obtenidos distasen mucho de los resultados esperados. A pesar de la limitación de cómputo produjo que surgieran nuevas técnicas como los **Algoritmos genéticos**, que buscaban mediante exploración y explotación el modelo óptimo que resolviese el problema. Si bien es cierto, la limitación de cómputo exigía dedicar miles de horas al programa para al final no obtener ningún resultado satisfactorio. Esto fue uno de los motivos de la cancelación de presupuestos por parte de los principales gobiernos (EEUU y Reino Unido) destinados a la investigación de la IA.

El siguiente gran *boom* de la IA llegó a mediados de los 70. Cuando el programa DENDRAL (1969), sistema para inferir la estructura molecular a partir de un espectrómetro de masas, se convirtió en el primer **Sistema experto**. El siguiente caso destacable de Sistema experto es MYCIN, destinado a análisis médicos y que a diferencia de DENDRAL, incluía incertidumbre en sus análisis. Estos sistemas eran efectivos por que se encontraban acotados a un dominio de conocimiento concreto y trabajaban con un reglas lógicas (450 aproximadamente el MYCIN), es decir, de respuesta verdadero o falso. Como resultado de estos éxitos se desarrolló el lenguaje de programación lógica **Prolog** en 1972. Durante esta época se presentó el concepto de **Lógica difusa** ó Lógica borrosa (del inglés *Fuzzy logic*) [González, 2011]. Definida en 1965 por Lofti A. Zadeh, esta técnica permite representar razonamientos de forma más similar a como razona un ser humano y su uso inicial consistió en sistemas de control de sistemas empotrados.

Desde entonces y a partir de los años 80, con computadores mucho más potentes que en los años 60, la IA se fue extendiendo poco a poco a áreas como por ejemplo: economía, escritura, videojuegos, noticias, conducción de vehículos y búsqueda de datos. Recientemente se han recuperado técnicas tales como Algoritmos genéticos y Redes de Neuronas, las cuales ya no se encuentran limitadas por el cómputo de las máquinas.

Durante los 60 también se trabajó en **búsqueda heurística**. Desarrollando algoritmos de búsqueda que se aprovechan de la información conocida para encontrar la solución óptima, como A*. Esto dio lugar a diferenciar entre Búsqueda informada (Búsqueda heurística) y Búsqueda no informada. La ventaja que presenta la Búsqueda heurística es que encuentra las soluciones óptimas (si existe) en menor tiempo que la búsqueda no informada. La búsqueda heurística tiene bastante importancia por su uso extendido en **búsqueda de caminos** o **planificación automática**. La planificación automática [Stuart y Russell, 2004] consiste en un proceso de búsqueda con el objetivo de encontrar una secuencia de acciones que resuelvan un problema concreto, un ejemplo de este tipo de problemas es el Problema del viajante [TSP, 1930], donde las acciones son los viajes entre las distintas ciudades.

Hoy en día, el principal problema al que se enfrenta la IA es el tratamiento de cantidades astronómicas de datos para la extracción de información útil, dando origen al nuevo campo del *Data Mining*.

A continuación se explicarán más en detalle técnicas mencionadas que tienen especial interés tanto en la robótica como en los videojuegos.

2.2.1.1. Máquinas de estados

La primera aproximación a la definición de una máquina de estados fue realizada por [McCulloch y Pitts, 1943], pero fueron Mealy [Mealy, 1955] y Moore [Moore, 1956] a través de sus aportaciones los que definieron lo que se conoce actualmente como máquina de estados. Su uso más extendido es el de compiladores y reconocedores de lenguajes. También se pueden utilizar para modelar comportamientos.

Las máquinas de estados se caracterizan por poseer los siguientes elementos:

- Estados: Definen una situación concreta y unívoca del entorno. Se debe definir un estado inicial, es decir, en qué estado se comienza cuando se ejecuta por primera vez. Si se desea que su ejecución finalice (Máquina de estado finito, MEF) se debe indicar un estado final.
- Transiciones: Elementos del entorno que provocan una transición de un estado a otro.
- Acciones: Comportamiento que debe ejecutarse cuando se transita o se está en un estado en concreto.

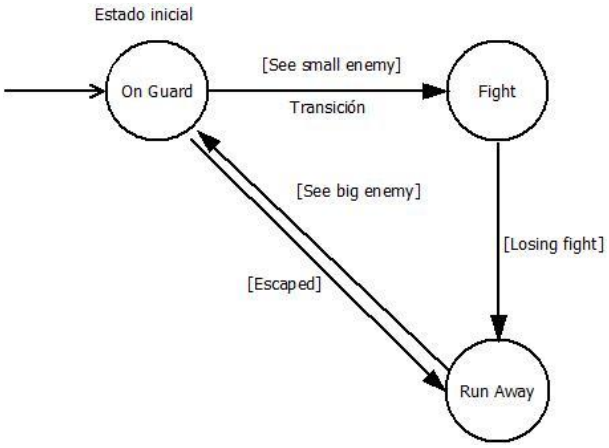


Ilustración 15 Máquina de estado aplicada a juegos de estrategia

Formalmente, las máquinas de estados se definen mediante una quíntupla que contiene los siguientes elementos:

- **Σ**, Alfabeto de entrada: datos de entrada para los estados, no puede ser vacío y debe ser un conjunto finito.
- **Q**, Conjunto de estados: no puede ser vacío. Se representan mediante círculos.
- $\boldsymbol{Q_0}$**,** Estado inicial: estado por el que comenzará a ejecutar. Se denota con un estado que recibe una transición que no procede de ningún otro estado.
- **f,** Tabla de transiciones: transiciones posibles entre estados.
- **F**, Estados finales: conjunto de estados en los que una vez se llega, se considera finalizada correctamente la ejecución. Se denotan con un círculo con doble borde.

Su ejecución comienza en el estado inicial (Q_0), y continúa hasta que llega a un estado final mediante transiciones. Si no existen estados finales (F), se ejecuta contínuamente.

Por ejemplo, en la Ilustración 15 Máquina de estado aplicada a juegos de estrategia:

- Alfabeto de entrada: [See small enemy], [Losing fight], [See big enemy] y [Escaped].
- Conjunto de estados: On Guard, Fight y Run Away.
- Estado inicial: On Guard.
- Tabla de transiciones:

Estado\Entrada	[See small enemy]	[Losing fight]	[See big enemy]	[Escaped]
On Guard	Ir a Fight	-	-	Ir a Run Away
Fight	-	Ir a Run Awat	-	-
Run Away	-	-	Ir a On Guard	-

Tabla 1 Tabla transiciones ejemplo MEF

- Estados finales: No existe ninguno.

Son sencillas de desarrollar y depurar (con comportamientos deterministas), pero cuando el número de estados y transiciones es muy elevado se vuelven muy complejas, por lo que son recomendables para dominios pequeños y concretos.

2.2.1.2. Árboles de comportamiento

Son un derivado de los Árboles de decisión. [Stuart y Russell, 2004] Un árbol consiste en un diagrama compuesto por nodos para la toma de decisiones, principalmente basado en condiciones booleanas. Hay tres tipos de nodos:

- Nodo raíz: Representa el comienzo del árbol y de él parten nodos internos u hojas.
- Nodo interno: Nodos que se encuentran entre la raíz y los nodos hoja. Representan condiciones adicionales que deben cumplirse. De un nodo interno pueden generarse más nodos internos o nodos hojas.
- Nodo hoja: Nodos terminales del árbol. Representan acciones. De un nodo hoja no se generan más nodos.

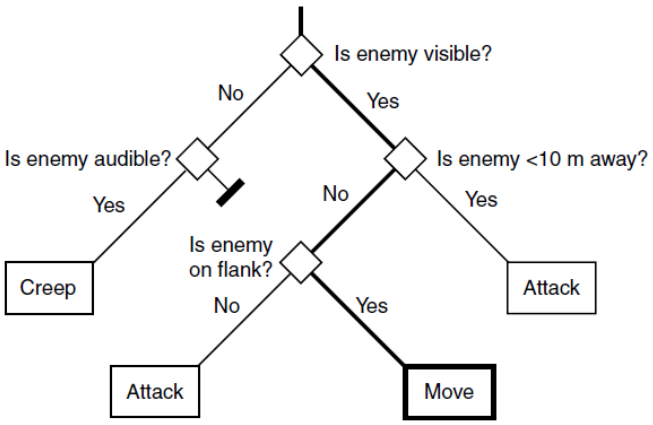


Ilustración 16 Ejemplo de Árbol de decisión

En el caso de los árboles de decisión, el orden para recorrer el árbol depende de las condiciones. Por ejemplo en la Ilustración 16, el nodo que verifica "Is enemy visible?" se corresponde con el nodo raíz. Los nodos "Is enemy audible?", "Is enemy <10 m away?" y "Is enemy on flank?" se corresponden con nodos intermedios ya que siguen evaluando condiciones y de ellos surgen nuevos nodos y los nodos "Creep", "Attack" y "Move" son nodos hoja ya que representan acciones y de ellos no surgen nuevos nodos.

En el caso de los Árboles de Comportamiento (del inglés, *Behaviour Tree* (BT)), se sigue una estructura de árbol similar a la de un árbol de decisión, donde se tiene un nodo raíz, nodos intermedios y nodos hojas. Se pueden utilizan para definir tanto

comportamientos sencillos (posible de implementar con una MEF) como comportamientos más complejos.

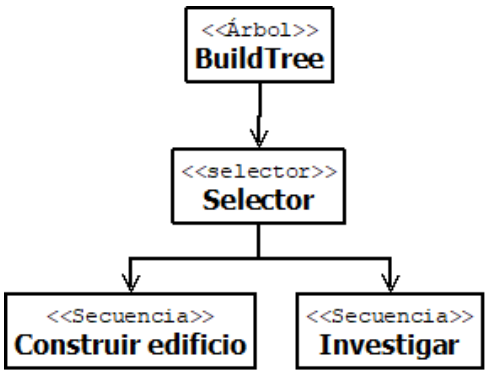


Ilustración 17 Ejemplo de Árbol de comportamiento

Un BT posee los siguientes elementos:

- Nodos hoja: Los nodos hojas comprueban o ejecutan acciones y devuelven un valor como resultado. En la mayoría de casos, el valor devuelto consiste en un estado indicando si ha tenido éxito, ha fallado, necesita más tiempo o ha ocurrido un error.
    - Condiciones: Comprueba que se cumplen una serie de condiciones y devuelve el estado correspondiente.
    - Acciones o Tareas: Cambia el estado del juego. Normalmente van a continuación de las condiciones.
- Nodos internos:
    - Compuestos (*composites*): Sirven para agrupar nodos y generar comportamientos más complejos.
        - Selectores: Recorren todos los nodos hijo hasta que alguno devuelva éxito, en cuyo caso, el selector devuelve éxito.
        - Secuencias: Recorren todos los nodos hijo y sólo devuelve éxito si todos los nodos hijos devuelven éxito.
    - Decoradores: Sirven para modificar el comportamiento de otros nodos internos, por ejemplo: repeticiones, condiciones extras o filtros. Un ejemplo de decorador sería limitar el número de veces que el bot puede ejecutar su comportamiento "Cobardía" que le hace esconderse detrás de obstáculos.

La principal ventaja que otorgan frente a los árboles de decisión es la posibilidad de definir comportamientos mucho más complejos. Esto se debe a la variedad de nodos existentes (secuencias, selectores y decoradores) que el árbol de decisión no posee. Adicionalmente, a diferencia de los árboles de decisión, donde su recorrido dependía de las condiciones, en un árbol de comportamiento su orden siempre es de arriba abajo y de izquierda a derecha, como se muestra en la

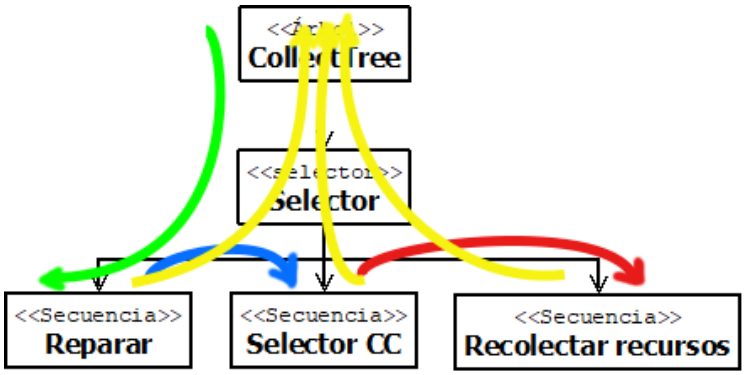


Ilustración 18 Orden de ejecución de un BT

Un ejemplo de ejecución, válido para la Ilustración 17, es: Inicialmente se ejecuta en el sentido de la línea verde, al ser *Selector* un nodo intermedio de tipo "Selector", en el caso de que la secuencia *Reparar* devuelva falso, pasaría a la secuencia inmediatamente a su derecha (línea azul), *Selector CC* y finalmente, en el caso de que *Selector CC* también falle (línea roja), se pasa a la última secuencia. En el caso de que cualquiera de las secuencias devolviese éxito, se seguiría el orden indicado por las líneas amarillas y finalizaría la ejecución del árbol.

2.2.1.3. Lógica difusa

Definida en 1965 por Lofti A. Zadeh. Surgió ante la necesidad de poder realizar razonamientos de forma más similar a la forma humana, pudiendo tener varios grados de certeza o falsedad. Hasta ese momento el conocimiento era verdadero o falso, no existían valores intermedios, por lo tanto, se podía dar el siguiente problema: Se quiere modelar un sistema de conducción automática de coches, y uno de sus elementos es el acelerador. Con los modelos hasta ese momento existentes, el sistema podría pisar el acelerador (a fondo) o no pisarlo. Como se puede deducir un sistema que pise a fondo el acelerador o no lo pise no es muy seguro, con la llegada de la lógica difusa se podían definir modelos que pisasen el acelerador en un rango de valores muy amplio y de una forma mucho más sencilla que con sistemas de verdadero o falso.

Por lo tanto, permite modelar comportamientos o situaciones, donde la división entre un estado u otro no es tan clara (pisar o no pisar el acelerador). Dichas divisiones están basadas en probabilidades y conocimiento experto para modelarse correctamente.

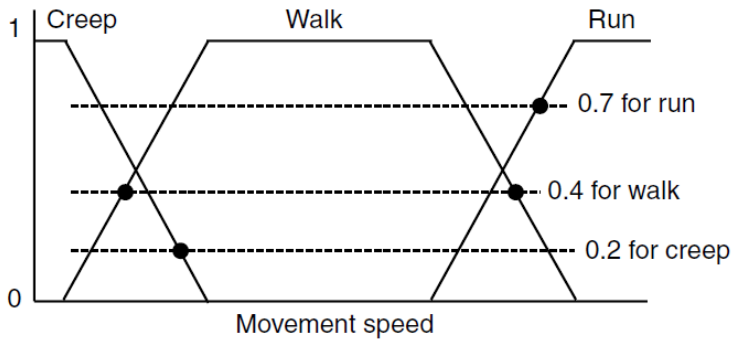


Ilustración 19 Lógica difusa aplicada a videojuegos

Está formado por:

- Conjuntos difusos: Funciones que sirven para conocer la pertenencia de la entrada a cada conjunto.
- Conjunto de reglas: Sirven para generar los conjuntos difusos.
- Valor lingüístico: Nombre de cada conjunto difuso.
- Función de pertenencia: Eje Y. Pertenencia de la entrada a cada conjunto difuso, es un valor entre 0 y 1.
- Variable lingüística: Eje X. Valor a evaluar.
- Motor de inferencia: Decide que reglas se activan cuando se recibe una entrada.

Su ejecución se puede dividir en dos fases:

- Borrosificación o *Fuzzyfication*: En esta fase la entrada se transforma a los valores de pertenencia de cada conjunto difuso.
- Desborrosificación o *Defuzzyfication*: Los conjuntos difusos formados en la *Fuzzyfication* se transforman a un único valor numérico que sirva como resultado.

La Ilustración 19 representa tres conjuntos difusos distintos (Creep, Walk y Run) que se han generado en base a una serie de reglas, la variable lingüística es Movement Speed y supongamos que el valor resultante indica el ruido que realiza un personaje al moverse.

El conjunto Creep se podría generar con las siguientes reglas:

Regla 1 (Creep): Si Movement Speed < 10
Entonces Creep = 1
Regla 2 (Creep): Si Movement Speed > 40
Entonces Creep = 0

Si queremos saber el ruido que se genera en el punto indicado como “0.4 for walk”, la función de pertenencia a cada conjunto difuso sería: 0 a Creep, 0.4 a Walk y 0.6 a Run, ya que la suma total debe dar 1. Se desborrosifican los valores y se obtiene un único valor numérico que es un valor entre 0 a 1 siendo 0 ningún ruido y 1 el máximo ruido posible.

Actualmente la lógica difusa se encuentra en multitud de aplicaciones, desde conducción automática pasando por cámaras de fotos, videojuegos o sistemas de control (tanto a nivel industrial como doméstico).

2.2.1.4. Arquitectura de pizarra

No se trata de una técnica de decisión en sí misma. Sirve para coordinar acciones entre distintos sistemas de control. Útil cuando el sistema está formado por varios sistemas que toman decisiones individualmente.

Está formado por los siguientes elementos, que se pueden visualizar gráficamente en la Ilustración 20:

- Expertos: Son los sistemas a coordinar. Cada experto propone la siguiente acción a realizar escribiéndola en la pizarra. Todos los expertos deben escribir en el mismo formato.
- Pizarra: Área donde todos los expertos pueden leer y escribir. Aquí se escriben las acciones propuestas.
- Árbitro: Es quien decide qué acción realizar en cada momento.

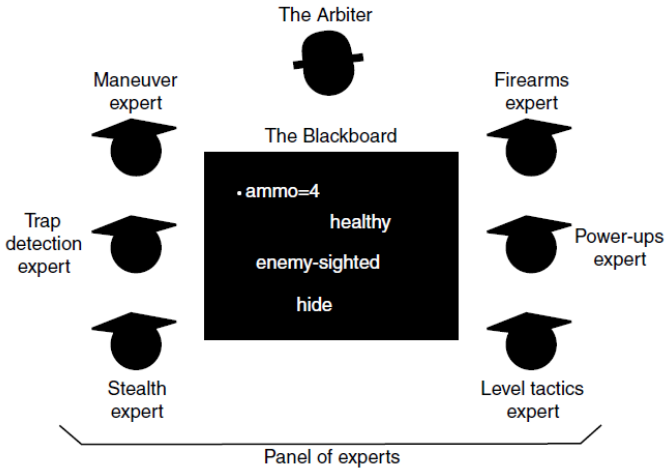


Ilustración 20 Arquitectura de pizarra ([Stuart y Russell,

A lo largo de la ejecución cada experto lee de la pizarra y escribe la acción que sugiere realizar. El árbitro evalúa todas las acciones basándose en una función de evaluación, normalmente por pesos o prioridades que asignan los propios expertos, y de todas las propuestas ejecuta la de mayor peso o prioridad.

La arquitectura de pizarra puede utilizarse en cualquier sistema con múltiples sistemas de decisión, tales como satélites [Corkill y Daniel, 1997] o robots [Unbehauen, 2009]

2.2.1.5. Redes de Neuronas

Definidas inicialmente por McCulloch y Pitts en 1943 [McCulloch y Pitts, 1943] con el perceptrón simple, surgieron a raíz de la intención de replicar estructuras naturales de forma matemática, por lo que su planteamiento está muy relacionado con las neuronas del cerebro. Sin embargo no ha sido hasta la época reciente cuando se han comenzado a utilizar de forma generalizada debido a su coste computacional.

Pueden utilizarse para casi cualquier aplicación, son fáciles de utilizar ya que no requieren una gran base matemática. Actualmente se utilizan sobre todo para reconocimiento de imágenes, texto o voz. Tienen como inconveniente que todos sus resultados y datos deben ser valores numéricos.

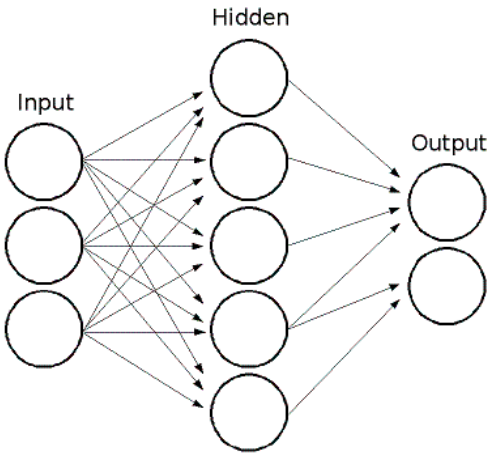


Ilustración 21 Ejemplo de red de neuronas (Fuente

Una red de neuronas está formada por los siguientes elementos:

- Neuronas: Es el elemento principal del sistema, recibe, procesa datos y están conectadas unas a otras. Cada neurona posee una función de activación que transforma la entrada recibida en otro valor numérico.
- Función de activación: Función matemática que transforma la entrada recibida de cada neurona. La función de activación correcta es fundamental para obtener la salida deseada.
- Pesos: Conexiones existentes entre cada neurona. Normalmente de izquierda a derecha, es decir, desde la entrada hacia la salida. Los pesos modifican los valores de salida de cada neurona. Pueden entenderse como la relevancia de cada neurona, por lo que entradas que no afecten al resultado, tendrán un peso 0.
- Capas: Las neuronas suelen agruparse en tres capas: Entrada, Oculta y Salida.
    - Capa de Entrada: Recibe como entrada los datos del problema y se conectan a las neuronas de la capa oculta, si existe, si no a las de salida.

- Capa Oculta: Puede existir una o más capas ocultas, todas ellas reciben como entrada la salida de otras neuronas y están conectadas a la siguiente capa o a la capa de salida.
- Capa de Salida: Conjunto de neuronas que reciben las salidas de otras neuronas y procesan los datos de forma que se obtiene un resultado.

Tras la definición inicial de red de neuronas artificiales han surgido multitud de variaciones que poseen estos elementos, siendo los más utilizados y conocidos actualmente:

- Adaline: Variación del perceptrón simple presentado por McCulloch y Pitts. Produce salida lineal y posee capa de entrada y de salida.
- Perceptrón Multicapa: Representado en la Ilustración 21. Se trata de una red de neuronas con una capa de entrada, al menos una capa oculta y una capa de salida. Su función de activación es no lineal.
- *Deep learning*: Variación del perceptrón multicapa que posee más de una capa oculta.

2.2.2. IA en los videojuegos

Todas las técnicas comentadas en 2.2.1. IA en la historia se han utilizado en mayor o menor medida según han ido evolucionando los videojuegos, las exigencias del jugador y la necesidad de innovar y destacar sobre la competencia. Al igual que con la IA, los primeros *NPCs* (Personaje No Jugador, del inglés, *Non-Player Character*) han evolucionado con el tiempo desde comportamientos más sencillos hasta comportamientos más complejos y elaborados, aunque, no por ello las técnicas más básicas han caído en desuso.

Podemos encontrar por ejemplo:

- **Máquinas de Estado** en juegos como *Pac-Man* o *Goldeneye 007* [PLG, 2016], donde el comportamiento de los fantasmas y de los enemigos respectivamente se modelaban con ME [Millington y Funge, 2009]. Principalmente esta técnica junto con los Árboles de Comportamiento suele ser la más usada para definir el comportamiento de los enemigos en juegos de disparos (o *First Person Shooter*, FPS), donde los comportamientos de los enemigos suelen ser: ver enemigo -> Cubrirse -> Atacar -> Vuelta a empezar. También se pueden utilizar para definir aspectos no relacionados con comportamientos de NPCs, como puede ser en *Guild Wars 2* donde las cadenas de eventos se pueden modelar mediante ME.
- **Árboles de Comportamiento** sustituyeron en gran medida las máquinas de estado, ya que son sencillos de manejar y permiten generar comportamientos más complejos de forma más fácil. Por ejemplo un comportamiento para cubrirse tras un obstáculo si hay enemigos visibles se puede definir mediante una única secuencia mientras que con una ME se

deben definir todos los estados y transiciones existentes. Al haber sustituido a muchas ME, su principal uso también se centra en los FPS, tales como *F.E.A.R*, *Killzone* o *Half Life*, aunque también se pueden encontrar en RPGs como *Dark Souls* o juegos de simulación y estrategia como *Spore* [Maxis, 2008].

- **Lógica difusa** se suele utilizar junto con otras técnicas como Redes de Neuronas o ME para conseguir buenos resultados [Pirovano, 2012]. Su uso principal consiste en control de comportamientos: movimiento, ataque, selección de armas… Los ejemplos más destacables son *Unreal* o *The Sims*.
- **Arquitectura de pizarra** su uso como organizador entre varios sistemas hace que su principal aplicación sea en videojuegos de estrategia, ya que existen distintos aspectos a controlar, por ejemplo: recursos, construcción, creación de unidades, movimiento, ataque o defensa. Aunque también tiene su uso extendido en FPS para coordinar varios NPCs a la vez, por ejemplo *No One Lives Forever 2*.
- **Redes de Neuronas** a diferencia del resto de técnicas, su uso es más reciente debido al coste computacional que implica. Suele utilizarse tanto para crear NPCs que mejoran a medida que se juega, *Forza Horizon 2* [Wired, 2014], como para definir comportamientos específicos, *Supreme Commander 2* [Mike Robbins, 2012].
- **Búsqueda heurística** tiene principal uso en la búsqueda de caminos, aunque se combina con otras técnicas como Waypoints [PLG, 2016] para reducir el espacio de búsqueda y hacerlo admisible. Cualquier videojuego con movimiento automático de NPCs utiliza búsqueda heurística, concretamente A*.
- **Planificación automática** su uso más extendido es en los juegos de estrategia, como StarCraft, donde se puede encontrar la secuencia de acciones más eficiente para por ejemplo construir una serie de edificio o entrenar unidades. [PLG, 2008]

2.3. Trabajos similares

Concretamente en el desarrollo de agentes para *StarCraft* hay distintas formas de enfocar el problema debido a la complejidad del juego. Se puede realizar una aproximación general como han realizado Weber, Mateas y Jhala [Weber et al., 2010], creando un sistema basado en Metas autodirigidas (del inglés *Goal Driven Autonomy* o GDA). La utilización de un GDA es acertada, ya que a lo largo de la partida los objetivos van cambiando, al principio queremos recursos, luego unidades militares, luego expandirnos… Sin embargo la creación de un sistema de este estilo puede ser muy compleja ya que requiere contemplar multitud de acciones y metas. En sus resultados han obtenido sobre un 75% de victorias contra la propia máquina, pero cerca de un 50% contra jugadores humanos, lo cual deja ver la dificultad del problema. En nuestro caso vamos a realizar un planteamiento

diferente, separando cada tarea (recolección, construcción, ataque, defensa, etc), de forma que haya un sistema específico encargado del mismo.

Partiendo de esta idea, algunas de estas aproximaciones ya realizadas han sido:

Identificación del orden óptimo para la construcción de edificios [Churchill y Buro, 2011], Churchill y Buro desarrollaron una búsqueda heurística basada en históricos de partidas jugadas para encontrar el mejor orden de construcción. El desarrollo de un componente exclusivo para definir órdenes de construcción es bastante efectivo, ya que si no se realiza correctamente puede perderse la partida. Sin embargo, al basarse en histórico de partidas (aunque sean de jugadores profesionales) sólo dispondrá de las órdenes de construcción más comunes. En nuestro caso para definir los órdenes de construcción se va a utilizar conocimiento experto con el objetivo de crear el orden más adecuado para el agente que se quiere desarrollar.

Por último, Synnaeve y Bessière [Synnaeve y Bessière, 2011a] [Synnaeve y Bessière, 2011b] han desarrollado dos modelos de Redes Bayesianas para dos aspectos distintos del juego, la primera, una red para predecir la estrategia inicial del enemigo, y la segunda una red para controlar las unidades militares.

- Identificar la estrategia inicial del enemigo: suele ser un factor decisivo muchas veces para poder ganar la partida, por ejemplo: si el enemigo se va a centrar en realizar una "Expansión temprana" un ataque rápido al comienzo de la partida puede dar la victoria en apenas unos minutos. Sin embargo, adivinar la estrategia al comienzo de la partida es complicado, por ello realizan varias predicciones, obteniendo a los 10 minutos de partida un acierto por encima del 94%.
- Controlar unidades militares individualmente este aspecto es casi igual de importante que el orden de construcción, ya que si se evalúa correctamente una situación o se ataca de forma adecuada un pequeño número de unidades puede contra otro mucho mayor. En este trabajo dividen los estados de una unidad en los cuatro más comunes: Scout (moverse y evitar daño), En Posición (Mantener la posición el máximo tiempo posible), Flock (Movimiento en grupo) y Lucha (Maximizar el daño), lo cual es una aproximación bastante buena y que con unidades de ataque a distancia consigue buenos resultados, sin embargo con unidades cuerpo a cuerpo el resultado es peor, ya que depende mucho de la unidad, no es lo mismo un fanático que un *zerling*. Por el contrario, estos buenos resultados se han obtenido en un entorno donde sólo existen unidades militares y no se tienen en cuenta el resto de aspectos del juego, por lo que al trasladarse al juego completo puede afectar al rendimiento ya que trata cada unidad de forma individual. Una aproximación más efectiva puede ser agrupar las unidades en grupo pequeños, de forma que sea más eficiente su manejo, e igualmente los resultados obtenidos deberían ser muy similares.

Estos dos trabajos tienen la estructura más parecida a la que se plantea en este trabajo, dividir las tareas dentro de cada juego y delegarlas a componentes específicos para su control. Aunque a diferencia de Redes Bayesianas, se utilizará Árboles de Comportamiento por que permitirán añadir y eliminar comportamientos de forma más fácil y sencilla.

2.4. StarCraft hoy en día

Con el paso de los años, los juegos de estrategia, tanto RTS como otros géneros se han convertido en entornos muy utilizados para el desarrollo y evaluación de técnicas de IA, ya que permiten en poco tiempo tener una beta del sistema.

En el caso de *StarCraft: Brood War* la herramienta más extendida hoy en día es la API de código BWAPI (2011) [BWAPI, 2017], escrito en C++ y de licencia y carácter *open source* y la herramienta *ChaosLauncher*, que permite utilizar *plugins* para *StarCraft: Brood War*, entre ellos un inyector de código para BWAPI. Este inyector lee y escribe en la región de memoria dedicada al programa, de esta forma se puede conectar el código del bot programado con BWAPI al juego. Gracias a este inyector y API, los usuarios pueden escribir agentes en lenguaje C++.

Al tratarse C++ de un lenguaje más cercano al bajo nivel que otros existentes, han surgido otras APIs, principalmente para Java, como JNI-BWAPI [JNI-BWAPI, 2015] ó BWMirror, que funcionan de envolturas (del inglés *wrapper*) para BWAPI, permitiendo crear agentes en otros lenguajes de programación.

A raíz de estas distintas APIs y la "facilidad" con la que se pueden generar agentes, han surgido torneos como SSCAIT [SSCAIT, 2017] o AIIDE [AIIDE, 2017], donde cualquiera puede mandar sus agentes, ya sea escrito en C++ o Java, y competir contra otros jugadores automáticos creados. Esto permite tanto a particulares como investigadores tener una plataforma grande y actualizada para evaluar sus agentes, ya que siempre siempre se están jugando partidas y quedan grabadas.

Recientemente Blizzard ha lanzado *StarCraft: Anthology* (Original + Expansión) de forma gratuita y cualquiera se lo puede descargar y jugar. El problema es que junto este lanzamiento también hay una nueva actualización, la 1.18 la cual no es compatible con la versión actual de BWAPI. Este hecho es sobre todo beneficioso de cara al futuro, ya que permitirá que cualquier persona, incluidas instituciones puedan utilizar *StarCraft* como plataforma para enseñar y crear IA. Es sólo cuestión de tiempo que la gente encargada de BWAPI lo haga compatible con la 1.18, dando así la libertad a cualquiera para crear un agente para *StarCraft*.

Por otro lado, Blizzard sacó en 2010 la continuación de *StarCraft*, *StarCraft* 2, manteniendo muchos aspectos del juego original y simplificando o cambiando otros, manteniendo la dificultad pero al mismo tiempo haciéndolo más accesible al jugador. Recientemente Google a través de su empresa de IA *Deep Mind* y en colaboración con Blizzard [Google, 2016] está desarrollando una API para

programar *bots* en *StarCraft* 2, lo cual si llega a la calidad y accesibilidad de BWAPI hará que tanto los aficionados como investigadores puedan elegir entre *StarCraft* y *StarCraft* 2 como entorno para sus investigaciones, aunque visto el apoyo que Blizzard ha dado a lo largo de los años a la entrega original, es poco probable que prescindan de los *bots* para *StarCraft 1*, siendo ahora gratuito.

Capítulo 3: Descripción del sistema

3.1. Introducción

En este capítulo se describe en detalle la arquitectura desarrollada. Está dividido en tres secciones principales: Estudio de alternativas, Análisis y Diseño.

En el Estudio de alternativas se muestran y evalúan distintas opciones existentes actualmente para el desarrollo de un agente para un RTS.

En la sección de Análisis se explica en detalle las características que posee el programa especificando sus casos de uso y requisitos.

Finalmente, en la sección de Diseño se realiza una descripción *top-bottom*, es decir, inicialmente se realiza una descripción general del sistema y se descompone posteriormente en elementos más pequeños.

3.2. Estudio de alternativas

En este apartado se analizan distintos aspectos relacionados con el diseño del agente: el juego, la técnica utilizada o la raza.

3.2.1. Entorno

Hoy en día existen diversos juegos de estrategia de carácter *open source* que tienen como clara referencias otros más clásicos, por ejemplo: *0 A.D.* [Wildfire Games, 2017] (*Age Of Empires*) u *OpenRA* [OpenRA, 2017] (*Command & Conquer*). No obstante, es más común la creación de motores *open source* que sustituyen el motor original del juego por otro más preparado para tecnologías actuales: *Wargus* [Wargus, 2011] (*Warcraft II*) u *openAge* [OpenAge, 2017] (*Age of Empires II*), que a su vez depende del juego original para mostrar gráficos y sonidos. Por otro lado, existen juegos como *StarCraft* que poseen un inyector de código que permite tener acceso a la IA del juego, aunque éstos son mucho menos comunes, ya que suelen violar las licencias de uso.

Cada uno posee sus ventajas, pero dado que para poder ejecutar *Wargus*, *openAge* y similares es necesario poseer el juego original, se van a analizar los juegos que son libres y originales, es decir *0 A.D.* y *OpenRA*, y *StarCraft*, ya que poseo una copia original y está disponible en la universidad.

0 A.D

Ilustración 22 Captura de 0 A.D. (Fuente

Juego totalmente libre, tanto la parte de código como la música, modelos o texturas. Está bastante actualizado, siendo su última versión estable de Noviembre de 2016. Las ventajas que proporciona utilizar este juego son:

+ Código libre y actualizado: Todas las ventajas e inconvenientes del software libre, como cambiar cualquier aspecto del juego, incluyendo la IA que es lo que buscamos.
+ JavaScript: Usar JavaScript como lenguaje de *scripting* tiene la ventaja de que es más sencillo frente a la gran mayoría de juegos que usan C++.
+ Donde quieras: Funciona tanto en sistemas operativos GNU/Unix como Windows o Mac.

Sin embargo, por el contrario encontramos unas desventajas que en el balance total hacen descartar 0 A.D. como juego donde implementar un agente.

– Sin experiencia de juego: Pese a ser muy similar a *Age of Empires* y *Age of Mythology*, juegos que jugué hace bastante y, a diferencia de *StarCraft*, conozco peor su funcionamiento interno: unidades, ataques, construcción…
– Sin entorno: A diferencia de *StarCraft* que ya tengo todo el entorno montado por la asignatura de "Inteligencia Artificial en la Industria del Entretenimiento", con *0 A.D.* habría que averiguar cómo conectar la IA al juego y cómo realizar pruebas para detectar errores.
– Sin experiencia con la API: No poseo ninguna experiencia usando la API disponible, lo que retrasaría todo el trabajo y como resultado podría salir un agente deficiente.

OpenRA

Ilustración 23 Captura de OpenRA - Tiberian Dawn (Fuente

A diferencia de 0 A.D. este se encuentra a medio camino entre juego libre y reimplementación del motor de juego, ya que permite jugar sin los juegos originales, pero si se poseen los juegos originales se usan todos sus aspectos artísticos: modelos, texturas, música… Su versión más actualizada es del 27 de Mayo de 2017.

Como ventajas si se seleccionase OpenRA como juego tendríamos:

+ Código libre y actualizado: Todas las ventajas e inconvenientes del software libre, como cambiar cualquier aspecto del juego, incluyendo la IA que es lo que buscamos.
+ Yaml: Usar Yaml para el *scripting* tiene la ventaja de que es más sencillo frente a cualquier lenguaje de programación y muchísimo más legible para humanos. Además el formato permite generar IAs de forma rápida y sencilla, lo que permite que sin mucho conocimiento se puedan generar una IA fácilmente.
+ Donde quieras: Funciona tanto en sistemas operativos GNU/Unix como Windows o Mac.

Como desventajas nos encontramos con los mismos problemas que *0 A.D.*:

– Sin experiencia de juego: Sin ningún tipo de experiencia en la saga *Command & Conquer* hace que primero sea necesario aprender a jugar, y aún así tampoco se podría aprender a jugar bien en poco tiempo, lo que llevaría a tener poco criterio para evaluar los resultados correctamente.
– *Scripting* demasiado sencillo: Tal y como están organizados los archivos correspondientes no se pueden definir comportamientos tan complejos como en *0 A.D.* o *StarCraft*.

En este caso la experiencia lo es todo y como el *scripting* es tan sencillo, es necesario saber jugar bien para poder generar comportamientos eficientes, y dado que no se posee ninguna experiencia, se descarta por completo.

StarCraft

Como ventajas a la hora de escoger *StarCraft* como entorno hay:

+ Experiencia: Se posee tanto experiencia con el juego (razas, estilos de juego, estrategias…) como con la API disponible para programar IAs.
+ Entorno: Se dispone de un entorno ya montando para programar y probar el código desarrollado.
+ IA: Existen APIs tanto en Java, C++ como Python. La programación de IAs es mucho más completa que en OpenRA, ya que permite definir cualquier comportamiento posible.

Como desventajas se encontrarían:

– IA: Exige definir cualquier comportamiento.
– Código privativo: Al tratarse de un juego comercial no se tiene acceso al código.
– No funciona en GNU/Unix: Sólo funciona en Windows.

En la elección de cada juego ha primado sobretodo la experiencia que se posee en el mismo, ya que no es posible desarrollar un buen jugador si no se sabe jugar. Esto se debe a la complejidad que poseen los juegos de estrategia, que pese a poder entenderse y aprender fácilmente las mecánicas básicas, se necesita de mucha práctica para aprender y reconocer las mejores técnicas. Por ejemplo, técnicas como el *Dropship* no se podría llevar a cabo correctamente si no se supiera en qué momento crear cada edificio, entrenar unidades y por cual zona del mapa atacar al enemigo.

No se han tenido en cuenta los recursos necesarios, ya que los tres juegos requieren muy pocos recursos para su ejecución.

3.2.2. Técnicas de IA

La decisión sobre qué técnica a utilizar se ha centrado principalmente en dos técnicas: Redes de Neuronas y Árboles de Comportamiento.

Una Arquitectura de Pizarra ha sido descartada debido a que no es necesario escoger una única acción a ejecutar en cada momento, se pueden ejecutar varias acciones de forma paralela. Lógica difusa se ha descartado porque sólo tendría utilidad en el ataque. Finalmente, una Máquina de Estado se ha descartado debido a la posibilidad de utilizar Árboles de Comportamiento, ya que son menos complejos.

Redes de Neuronas

Técnica bastante fácil de usar y de la que existen multitud de implementaciones en cualquier lenguaje de programación. Las ventajas se encontrarían:

+ Fácil de usar: No requieren mucho conocimiento específico ni matemático.
+ Buenos resultados: Dan buenos resultados en casi cualquier entorno.

Como desventajas se encontrarían:

– Numéricas: Todas las entradas y salidas deben ser valores numéricos, por lo que es necesario codificar en número todo aquello que no lo sea, como por ejemplo, los edificios.
– Ejemplos: Se necesitan ejemplos codificados de partidas anteriores para poder entrenar la red.
– Entrenamiento: Es necesario entrenar la red cada vez que quieren realizar cambios en la misma.

Árbol de Comportamiento

Técnica bastante sencilla de implementar y utilizar. Es bastante usada en videojuegos de diversos tipos. Las ventajas que se encontrarían son:

+ Fácil de implementar: Es más sencillo implementar un árbol de comportamiento que un modelo de red de neuronas.
+ Fácil de usar: Al no depender de números su uso es más sencillo y permite realizar modelos de forma gráficas.
+ Escalabilidad: Es muy escalable, permitiendo generar comportamientos complejos de forma sencilla.

Como desventajas se encontraría:

– Escalabilidad: Comportamientos muy complejos vuelven al árbol difícil de visualizar si no es de forma gráfica.

El principal motivo por el que no se han escogido las Redes de Neuronas y sí los Árboles de Comportamiento son estas dos características que no necesita un árbol de comportamiento:

- Valores numéricos: es muy complicado codificar todas las entradas del juego como valores numéricos, ya que elementos como *choke points* o expansiones no son fácilmente representables de forma numérica. Un árbol de comportamiento no posee esta limitación.
- Resultados: Para obtener resultados se deben asignar valores a sus atributos (número de neuronas, función de activación, número de capas…), entrenar la red, y una vez entrenada no se sabe a ciencia cierta cuándo ni cómo se realizan ciertas acciones. Por el contrario, en un árbol de

comportamiento se sabe qué comportamientos están definido y en qué orden y situación se va a ejecutar de forma clara.

3.2.3. Raza

La selección de la raza *Terran* como raza del agente se debe principalmente a las siguientes características personales:

- Experiencia: Mi experiencia de juego con la raza *Terran* es muchísimo mayor que con las otras dos razas. Lo que lleva a conocerla mejor y obtener como consecuencia mejores resultados.
- Construcción: A diferencia de los *Protoss* o *Zerg*, pueden construir donde quieran ya que no depende de un campo de energía o biomateria, respectivamente.
- Agente: El agente que ya tenía desarrollado parcialmente de la asignatura "Inteligencia Artificial en la Industria del Entretenimiento" era de la raza *Terran*, por lo que el trabajo era menor que empezar desde cero.

3.3. Análisis del sistema

En este apartado se habla sobre las características del programa desarrollado, especificando restricciones del sistema, detallando su entorno operacional y especificando tanto sus casos de uso como requisitos existentes.

3.3.1. Descripción de las características funcionales

Como se comentó en 1.3 Objetivos del trabajo, el agente desarrolladora deberá ser capaz de ganar a la IA del propio juego, para ello deberá ser capaz de realizar acciones del juego tales como:

- Entrenar unidades, tanto trabajadores como unidades militares.
- Construir edificios. Y en menor medida, ser capaz de repararlos si son dañados.
- Controlar las unidades militares para atacar o defender.

Estas son las tres acciones principales en las que se puede agrupar el juego y por tanto, se incluyen en la implementación del agente. Adicionalmente, deberá:

- Crear un mapa de influencia para las tácticas de ataque/defensa.

Existen también las siguientes limitaciones en el agente desarrollado:

- Sólo funciona para la raza *Terran*.
- Está pensado para jugar contra un único jugador. Se puede utilizar en partidas de más de 2 jugadores, pero no se asegura su correcto funcionamiento.
- No puede cargar y descargar unidades en vehículos como la "Nave de evacuación" por lo que en mapas que depende de estas técnicas no podrá ganar nunca.

3.3.2. Restricciones del sistema

Los requisitos mínimos para poder ejecutar correctamente este trabajo a nivel de hardware son:

- RAM: 4 GB.
- Procesador: 2 GHz.
- Tarjeta Gráfica: Tarjeta gráfica SVGA compatible con DirectX o superior.

Los requisitos a nivel de software son:

- Sistema operativo: Debe ser una versión de Windows compatible con *StarCraft: Brood War*, Java 8 y BWAPI 3.7.5. Puede ser de 32 o 64 bits. Algunos ejemplos de sistemas compatibles son: Windows 7, Windows 8 y Windows 10.
- El equipo debe tener instalado *StarCraft: Brood War* con la versión 1.16.1
- El equipo debe tener instalado la versión 3.7.5 de BWAPI.
- El equipo debe tener instalado *ChaosLauncher* en la versión 0.5.2, que es compatible con la 1.16.1 de *StarCraft*.
- El equipo debe tener instalado Java 8 de 32 bits.

3.3.3. Entorno operacional

El entorno utilizado para el agente consta principalmente de cuatro componentes: *StarCraft: Brood War*, BWAPI, *ChaosLauncher* y JNI-BWAPI.

- BWAPI se puede encontrar en el siguiente enlace: https://github.com/bwapi/bwapi/releases
- En caso de que ChaosLauncher no vaya incluido con BWAPI se puede encontrar en el siguiente enlace: http://www.teamliquid.net/forum/brood-war/65196-chaoslauncher-for-1161
- JNI-BWAPI se puede encontrar en el siguiente enlace: https://github.com/JNI-BWAPI/JNI-BWAPI

Como se ha comentado, ***StarCraft: Brood War*** será el entorno donde se ejecute el agente desarrollado. Para el correcto funcionamiento, deberá utilizarse la versión 1.16.1.

Para el desarrollo y ejecución del agente se utiliza la librería BWAPI, el *wrapper* de Java, JNI-BWAPI y el IDE de desarrollo Eclipse.

Eclipse es una IDE (del inglés, *Integrated Development Environment*) pensado principalmente para el lenguaje Java, aunque también existe versión para C y C++. Esta herramienta se ha utilizado para el desarrollo del código realizado y generación del .jar resultante. La Ilustración 24 muestra una captura del IDE. Se puede encontrar para descargar en el siguiente enlace: https://eclipse.org/

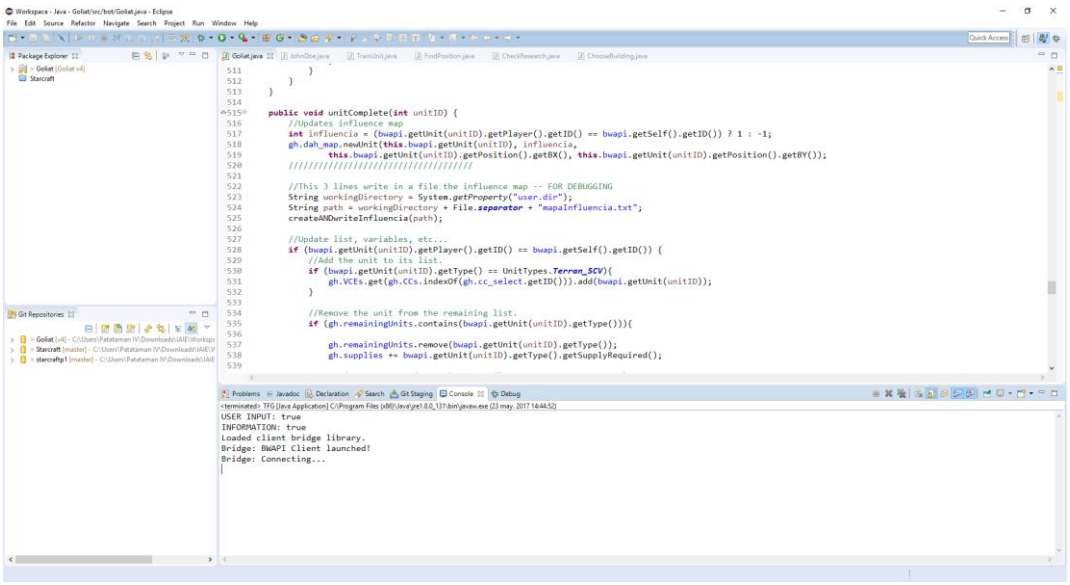


Ilustración 24 Eclipse

BWAPI [BWAPI, 2017] es un API o*pen source* escrita en C++ que permite programar *bots* en el mismo lenguaje para StarCraft: Brood War. A través de una serie de llamadas a la API se disparan automáticamente los eventos del juego que permiten: atacar, moverse, construir, entrenar, comprobar atributos de las unidades (como tipo o puntos de vida), comprobaciones del terreno (altura o si es construible)… Posee una serie de eventos escucha (del inglés, *listener*) que son llamados en distintas ocasiones a lo largo del juego y que nos dan información relevante sobre el entorno y el desarrollo de la partida:

- *onStart:* Evento que se ejecuta nada más empezar la partida.
- *onEnd:* Evento que se ejecuta al finalizar la partida.
- *onFrame:* Evento que se ejecuta en cada *frame* del juego.
- *onSendText:* Evento que se ejecuta cuando el usuario escribe un mensaje dentro del juego.
- *onReceiveText:* Evento que se ejecuta cuando se recibe texto de otro jugador.
- *onPlayerLeft:* Evento que se ejecuta cuando un jugador deja la partida.
- *onNukeDetect:* Evento que se ejecuta cuando se lanza una bomba nuclear.
- *onUnitDiscover:* Evento que se dispara cuando una unidad sale de la niebla de guerra.
- *onUnitEvade:* Evento que se dispara cuando una unidad es ocultada por la niebla de guerra.
- *onUnitShow:* Evento que se llama cuando una unidad invisible pasa a ser visible.
- *onUnitHide:* Evento que se llama cuando una unidad pasa de visible a invisible.
- *onUnitCreate:* Evento que se llama cuando una unidad se crea.
- *onUnitDestroy:* Evento que se llama cuando una unidad se destruye.

- *onUnitMorph*: Evento que se llama cuando una unidad cambia su tipo.
- *onUnitRenegade*: Evento que se llama cuando una unidad cambia de dueño.
- *onSaveGame*: Evento que se llama al guardar la partida.
- *onUnitComplete*: Evento que se llama cuando una unidad cambia su estado de "incompleta" a "completa".

A pesar de todas estas funciones, BWAPI no se posee ninguno que alerte cuando una unidad está siendo atacada, lo que significa que para conocer esa información deben realizarse comprobaciones en cada iteración del juego.

Por otro lado, la API por sí sola no puede ejecutar los *bots* desarrollados en el juego. Para ello se necesita del programa *ChaosLauncher*.

ChaosLauncher (Ilustración 25) es una herramienta que permite utilizar aplicaciones de terceros junto con *StarCraft*. Las aplicaciones (o *plugins*) más comunes son:

- Inyector de código: Es el *plugin* que nos permitirá conectar nuestro agente desarrollado usando BWAPI con *StarCraft*.
- *W-Mode*: Permite ejecutar el juego en modo ventana.
- *APMAlert*: *Plugin* que contabiliza las acciones por minuto realizadas y alerta si bajan de cierto número.
- ICCup: *Plugin* que monitoriza para evitar que se realicen trampas durante la partida.

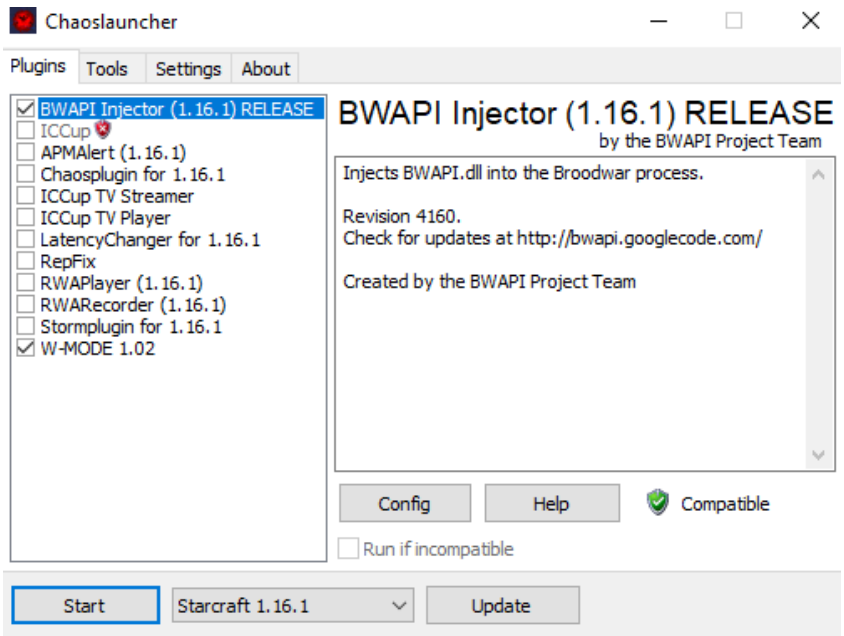


Ilustración 25 ChaosLauncher

Por último, **JNI-BWAPI** consiste en un *wrapper* de la versión 3.7.4 de BWAPI. Fue desarrollado utilizando la *Java Native Interface* (JNI) y permite programar el agente en Java utilizando llamadas muy similares a las existentes en BWAPI.

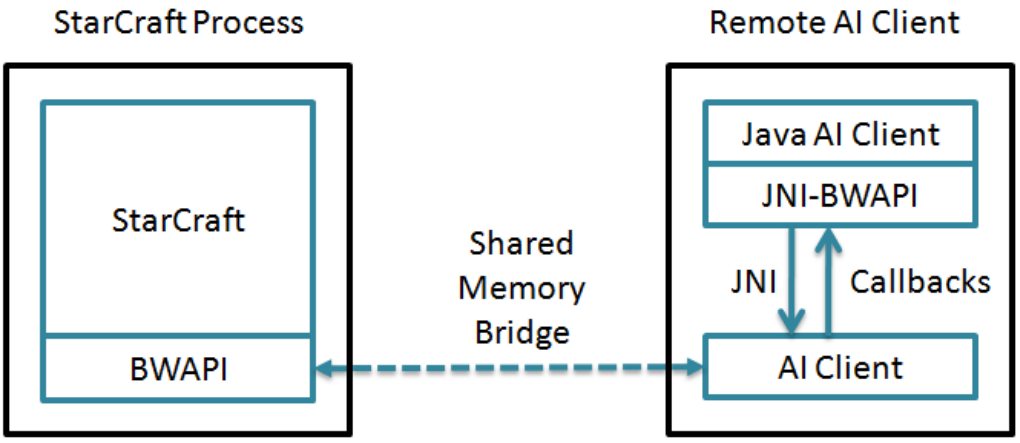


Ilustración 26 Esquema de JNI-BWAPI (Fuente

La Ilustración 26 representa un diagrama sobre el funcionamiento de JNI-BWAPI. Funciona como si él mismo fuese un bot escrito en C++ que se comunica con BWAPI (AI Cliente), esto permite aprovechar la región de memoria compartida entre el *bot* y la API en C++ reduciendo tiempos de ejecución. Además gracias a la JNI pueden realizarse llamadas entre el programa en Java y la API en C++ lo cual permite tener las mismas funciones (no con el mismo nombre) que en la API original. El problema existente con esta API es que está descontinuada ya que no se actualiza desde hace tres años, por lo que no está adaptada a las versiones más recientes de BWAPI, como la 4.2.0, y hace que sólo se pueda utilizar con las versiones 3.7.4 y la 3.7.5, que soluciona fallos de la versión anterior.

3.3.4. Especificación de casos de uso

En este apartado se especifican los casos de uso. Los casos de uso representarán de forma global las acciones a realizar por los actores que pueden interactuar con el juego (usuario y agente), y en base a estos casos de uso, posteriormente se extraerán los requisitos que ayudarán a diseñar el sistema.

3.3.4.1. Descripción de los actores

Los actores presentes que interactúan con el juego son dos: El Usuario y el Agente desarrollado.

- El Usuario realiza todas las acciones del juego no relacionadas con jugar la partida, esto abarca principalmente: menús y ajustes.
- El Agente interactúa con el juego mediante todas las acciones relacionadas con jugar la partida.

3.3.4.2. Descripción de los atributos de los casos de uso

Código	
Nombre	
Actores	
Descripción	
Precondiciones	
Postcondiciones	
Escenario	
Escenario de error	

Tabla 2 Ejemplo Tabla Caso de uso

- **Código:** Representa el identificador único para el caso de uso. Seguirá el esquema: CU-XX, siendo XX dos número entre el 0 y el 9, por ejemplo CU-13.
- **Nombre:** Identificación extendida del caso de uso. Servirá para describir muy brevemente el caso de uso.
- **Actores:** Entidades que participan en el caso de uso.
- **Descripción:** Objetivo que se busca alcanzar con la acción realizada en el caso de uso.
- **Precondiciones:** Circunstancias que han de producirse antes de la realización de las acciones que componen el caso de uso.
- **Postcondiciones:** Circunstancias que una vez completado el caso de uso se dan en el sistema.
- **Escenario:** Conjunto de pasos que han de darse para poder pasar de las precondiciones a las postcondiciones.
- **Escenario de error:** Pasos o situaciones que si ocurren, la ejecución puede derivar en un error.

3.3.4.3. Lista de casos de uso

Código	CU-01
Nombre	Iniciar juego
Actores	Usuario
Descripción	El usuario inicia *StarCraft: Brood War* a través de *ChaosLauncher* para posteriormente poner al agente a jugar
Precondiciones	• Tener *StarCraft: Brood War* instalado y en la versión 1.16.1 • Tener *ChaosLauncher* con el *plugin* del inyector de código instalado.
Postcondiciones	• Se ejecuta *StarCraft: Brood War*.
Escenario	1. Se abre *ChaosLauncher* y se selecciona el inyector de código. 2. Se pulsa "*Start*" o "Iniciar" 3. Se selecciona "Un jugador" y se escoge la opción "Expansión".
Escenario de error	1. Se selecciona "Un jugador" y se escoge la opción "Original". 2. No se inicia *StarCraft* a través de *ChaosLauncher*.

Tabla 3 CU-01 Iniciar juego

Código	CU-02
Nombre	Iniciar agente
Actores	Usuario
Descripción	El usuario inicia el agente desarrollado para que esté a la espera de la inicialización de una partida.
Precondiciones	• Tener BWAPI 3.7.4 o 3.7.5 instalado. • Tener Java 8 instalado. • Tener el .jar y las librerías .dll en la misma carpeta.
Postcondiciones	• El agente se encuentra inicializado a la espera de partida.
Escenario	1. Se abre una consola de comandos. 2. Desde la consola se va a la carpeta donde se encuentra el .jar del agente. 3. Se ejecuta introduciendo el comando correcto.
Escenario de error	1. No se tienen en la misma carpeta el .jar y los .dll. 2. No se introduce el comando correctamente.

Tabla 4 CU-02 Iniciar agente

Código	CU-03
Nombre	Crear partida
Actores	Usuario
Descripción	El usuario crea una partida para que juegue el agente.
Precondiciones	• Tener el agente iniciado. • Tener *StarCraft: Brood War* iniciado.
Postcondiciones	• Se inicia una partida donde juega el agente.
Escenario	1. El usuario selecciona para jugar contra la máquina. 2. Selecciona un mapa cualquiera. 3. Elige como raza del jugador, a los Terran. 4. Le da a "Aceptar".
Escenario de error	1. No se selecciona la raza Terran como raza del jugador.

Tabla 5 CU-03 Crear partida

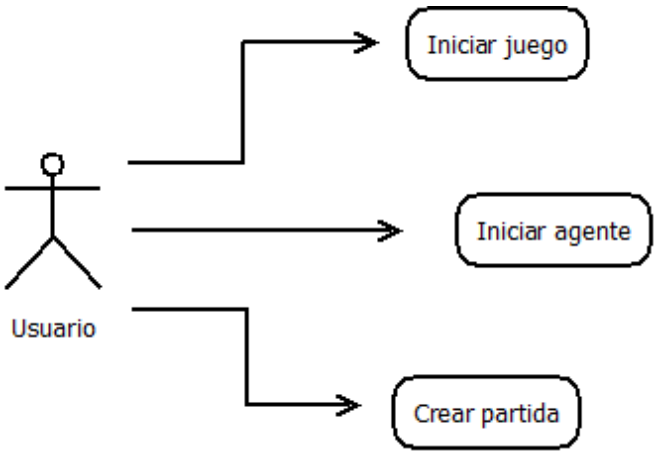


Ilustración 27 Diagrama casos de uso Usuario

Código	CU-04
Nombre	Entrenar unidad
Actores	Agente
Descripción	El agente entrena una nueva unidad.
Precondiciones	• Tener los recursos necesarios para crear la unidad. • Tener finalizado el edificio correspondiente. • Se deben tener suministros suficientes. • El edificio no debe estar ocupado.
Postcondiciones	• Se comienza a entrenar una nueva unidad.
Escenario	1. El agente comprueba que en el momento actual se tengan recursos suficientes. 2. Se busca el edificio correspondiente. 3. Si está libre se entrena una nueva unidad.
Escenario de error	1. En el transcurso entre comprobar los recursos y poner a entrenar la unidad ya no se poseen los recursos suficientes.

Tabla 6 CU-04 Entrenar unidades

Código	CU-05
Nombre	Construir edificio
Actores	Agente
Descripción	El agente construye nuevos edificios.
Precondiciones	• Tener recursos suficientes. • La localización debe ser válida. • Debe existir un camino para poder llegar a la localización.
Postcondiciones	• Se comienza a construir un nuevo edificio.
Escenario	1. El agente comprueba que existen trabajadores libres. 2. Se selecciona a uno de los trabajadores libres. 3. Se comprueba que se poseen recursos suficientes. 4. Se busca y encuentra una posición válida para construir el edificio. 5. Se manda construir el edificio.
Escenario de error	1. En el transcurso entre comprobar los recursos y construir el edificio ya no se poseen los recursos suficientes. 2. En el transcurso entre comprobar los recursos y construir el edificio la posición se ha vuelto inválida.

Tabla 7 CU-05 Construir edificios

Código	CU-06
Nombre	Atacar
Actores	Agente
Descripción	El agente ordena a un grupo de unidades atacar una posición.
Precondiciones	• Tener unidades militares. • Obtener una posición de ataque accesible. • Las unidades militares deben estar disponibles.
Postcondiciones	• Se ordena a un grupo de unidades atacar una posición concreta.
Escenario	1. Se comprueba que se tienen unidades suficientes para atacar. 2. Se comprueba el estado de las unidades. 3. Se obtiene una posición de ataque. 4. Se les ordena atacar la posición.
Escenario de error	1. En el transcurso entre comprobar las unidades y ordenarles atacar, han muerto y no se han actualizado bien los datos. 2. En el transcurso entre comprobar las unidades y ordenarles atacar, su estado ha cambiado y ya no es válido el ataque.

Tabla 8 CU-06 Atacar

Código	CU-07
Nombre	Defender
Actores	Agente
Descripción	El agente realiza acciones defensivas utilizando unidades militares.
Precondiciones	• Poseen unidades militares.

Postcondiciones	• Pueden darse dos situaciones:
	  ○ Las unidades militares se introducen en un búnker.
	  ○ Las unidades militares se mueven a una posición accesible determinada.

Escenario	1. Se selecciona una unidad militar que no esté defendiendo. 2. Dos posibles opciones: 2.1. Si existen búnkeres vacios, se le ordena ir al bunker. 2.2. Si no existen búnkeres o están llenos, se le ordena defender una zona específica.
Escenario de error	1. En el transcurso entre comprobar las unidades y ordenarles ir a un búnker, el bunker es destruido.

Tabla 9 CU-07 Defender

Código	CU-08
Nombre	Influencia
Actores	Agente
Descripción	El agente actualiza el mapa de influencias.
Precondiciones	• Tener inicializado el mapa de influencias. • Se finaliza la creación de una unidad o edificio.
Postcondiciones	• El mapa de influencias se actualiza con los nuevos valores.
Escenario	1. Se finaliza la creación de una unidad o edificio. 2. Se identifica la unidad y se aplica la influencia correspondiente en el mapa.
Escenario de error	1. Por un error de la API, no se reconoce la creación de la unidad o edificio. 2. Por un error de la API se considera que una unidad se ha completado más de una vez. 3. Por funcionamiento de la API se considera completada la unidad o edificio antes de que realmente esté finalizado.

Tabla 10 CU-08 Influencia

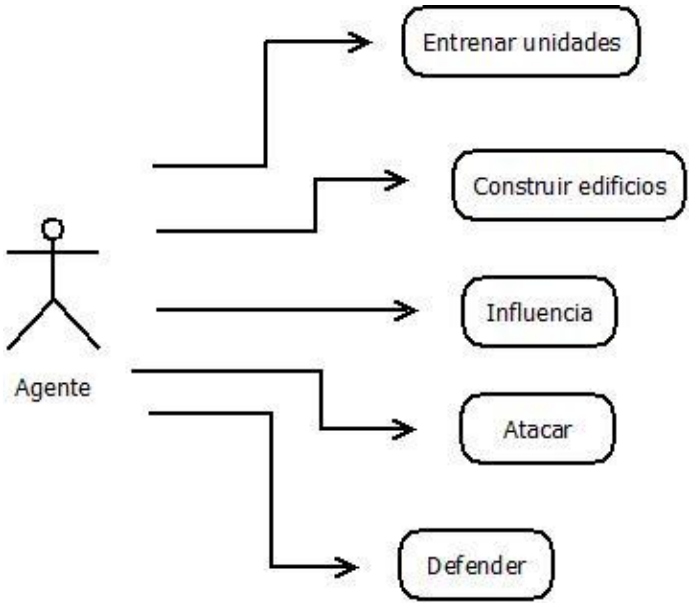


Ilustración 28 Diagrama casos de uso Agente

3.3.5. Especificación de requisitos

En este apartado se van a especificar los requisitos extraídos a partir de los casos de uso definidos en 3.3.4. Especificación de casos de uso.

3.3.5.1. Descripción de los atributos de los requisitos

Código		**Fuente**	
Título			
Descripción			
Necesidad		**Prioridad**	
Estabilidad		**Verificabilidad**	

Tabla 11 Tabla ejemplo de requisito

- **Código:** Identificador único del requisito. Existen dos tipos de requisitos: requisitos funcionales, referidos a características que deberá cumplir el sistema, y requisitos no funcionales, referidos a restricciones que deberá cumplir el sistema. Los requisitos funcionales seguirán el esquema: RF-YY-XX. Los requisitos no funcionales seguirán el esquema RNF-YY-XX, siendo YY en ambos casos el número de Caso de Uso correspondiente y XX dos número entre el 0 y el 9, por ejemplo RF-01-13 y RNF-05-03. En caso de ser extraído de más de un caso de uso, se usa el identificador más pequeño.
- **Título:** Breve descripción del requisito.
- **Descripción:** Descripción detallada del requisito.
- **Fuente:** Caso de uso del que ha sido extraído el requisito.

- **Necesidad:** Determina el grado de implementación del requisito.
  - Esencial: Es obligatorio que el requisito esté implementado.
  - Deseable: Es bastante recomendable que el requisito se encuentre implementado.
  - Opcional: El requisito se podrá implementar, pero no es obligatorio.
- **Prioridad:** Indica cómo de importante es el requisito en la implementación final, permitiendo establecer así un orden de implementación.
  - Alta: El requisito debe ser de los primeros en implementarse.
  - Media: El requisito debe ser implementado una vez se han implementado los de prioridad alta.
  - Baja: El requisito debe ser implementado al final del desarrollo.
- **Estabilidad:** Indica si el requisito puede ser modificado durante el desarrollo.
  - Estable: El requisito no puede variar.
  - Inestable: El requisito puede variar.
- **Verificabilidad:** Indica el grado con el que es posible comprobar que el requisito ha sido implementado.
  - Alta: Es fácil de comprobar que el requisito ha sido implementado. No hay necesidad de acceder a código para verificarlo.
  - Media: Se puede comprobar que el requisito ha sido implementado. Es complicado de verificar sin acceder al código.
  - Baja: Es difícil comprobar que el requisito ha sido implementado, tanto revisando el código como ejecutando el sistema.

3.3.5.2. Lista de requisitos

Para facilitar la organización de los requisitos se han organizado siguiendo el siguiente orden: primero los requisitos no funcionales y posteriormente los requisitos funcionales. Éstos a su vez, están organizados internamente en orden ascendente de identificador de fuente de Caso de Uso.

Código	RNF-01-01	**Fuente**	CU-01 Iniciar juego
Título	Sistema Operativo		
Descripción	El agente deberá ser desarrollado para Windows 7, 8 o 10.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 12 Requisito RNF-01-01 Sistema Operativo

Código	RNF-01-02	**Fuente**	CU-01 Iniciar juego
Título	Entorno		
Descripción	El agente deberá ser desarrollado para *StarCraft: Brood War* en la versión 1.16.1		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 13 Requisito RNF-01-02 Entorno

Código	RNF-01-03	**Fuente**	CU-01 Iniciar juego
Título	*ChaosLauncher*		
Descripción	El inyector de código de *ChaosLauncher* debe ser para la versión 1.16.1 de *StarCraft*.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 14 Requisito RNF-01-03 ChaosLauncher

Código	RNF-02-01	**Fuente**	CU-02 Iniciar agente
Título	API		
Descripción	El agente deberá ser desarrollado con la API JNI-BWAPI		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Media

Tabla 15 Requisito RNF-02-01 JNI-BWAPI

Código	RNF-02-02	**Fuente**	CU-02 Iniciar agente
Título	Lenguaje		
Descripción	El agente deberá desarrollarse en Java 8.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Media

Tabla 16 Requisito RNF-02-02 Lenguaje

Código	RNF-02-03	**Fuente**	CU-02 Iniciar agente
Título	BWAPI		
Descripción	La versión instalada de BWAPI debe ser la 3.7.4 o 3.7.5		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 17 Requisito RNF-02-03 BWAPI

Código	RNF-02-03	**Fuente**	CU-02 Iniciar agente
Título	Raza		
Descripción	La raza del agente desarrollado deberá ser *Terran*		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 18 Requisito RNF-02-04 Raza

Código	RNF-06-01	**Fuente**	CU-06 Atacar
Título	Enemigo		
Descripción	El agente se podrá enfrentar a cualquiera de las tres razas.		
Necesidad	Esencial	**Prioridad**	Media
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 19 Requisito RNF-06-01 Enemigo

Código	RF-04-01	**Fuente**	CU-04 Entrenar unidades CU-05 Construir edificios
Título	Recursos		
Descripción	El agente debe ser capaz de recolectar recursos (minerales y gas vespeno) para la raza *Terran*.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 20 Requisito RF-04-01 Recursos

Código	RF-04-02	**Fuente**	CU-04 Entrenar unidades CU-05 Construir edificios
Título	Entrenar		
Descripción	El agente debe ser capaz de entrenar nuevas unidades para la raza *Terran*.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 21 Requisito RF-04-02 Entrenar

Código	RF-05-01	**Fuente**	CU-05 Construir edificios
Título	Construir		
Descripción	El agente debe ser capaz de construir nuevos edificios para la raza *Terran*.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 22 Requisito RF-05-01 Construir

Código	RF-05-02	**Fuente**	CU-05 Construir edificios
Título	Mapa construcción		
Descripción	El agente debe generar un mapa con las posiciones válidas para construir para la raza *Terran*.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Media

Tabla 23 Requisito RF-05-02 Mapa construcción

Código	RF-05-03	**Fuente**	CU-05 Construir edificios
Título	Evaluar terreno		
Descripción	El agente debe ser capaz de variar las localizaciones de construcción favoreciéndose del terreno para la raza *Terran*.		
Necesidad	Deseable	**Prioridad**	Media
Estabilidad	Estable	**Verificabilidad**	Media

Tabla 24 Requisito RF-05-03 Evaluar terreno

Código	RF-05-04	**Fuente**	CU-05 Construir edificios CU-07 Defender
Título	*Choke Points*		
Descripción	El agente debe ser capaz de identificar los Choke Points del mapa para la raza *Terran.*		
Necesidad	Deseable	**Prioridad**	Media
Estabilidad	Estable	**Verificabilidad**	Media

Tabla 25 Requisito RF-05-04 Choke Points

Código	RF-05-05	**Fuente**	CU-05 Construir edificios
Título	Reparar		
Descripción	El agente debe ser capaz de identificar reparar edificios dañados para la raza *Terran*.		
Necesidad	Opcional	**Prioridad**	Baja
Estabilidad	Inestable	**Verificabilidad**	Media

Tabla 26 Requisito RF-05-05 Reparar

Código	RF-05-06	**Fuente**	CU-05 Construir edificios
Título	Añadidos		
Descripción	El agente debe ser capaz de construir añadidos en los edificios que lo requieran para la raza *Terran*.		
Necesidad	Deseable	**Prioridad**	Baja
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 27 Requisito RF-05-06 Añadidos

Código	RF-05-07	**Fuente**	CU-05 Construir edificios
Título	Expansión		
Descripción	El agente debe ser capaz de expandirse para la raza *Terran*.		
Necesidad	Esencial	**Prioridad**	Media
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 28 Requisito RF-05-07 Expansión

Código	RF-05-08	**Fuente**	CU-05 Construir edificios CU-07 Defender
Título	Búnkeres		
Descripción	El agente debe ser capaz de construir búnkeres en zona críticas como los *Choke Points* o zonas de baja influencia para la raza *Terran*.		
Necesidad	Deseable	**Prioridad**	Media
Estabilidad	Estable	**Verificabilidad**	Media

Tabla 29 Requisito RF-05-08 Búnkeres

Código	RF-06-01	Fuente	CU-06 Atacar CU-07 Defender CU-08 Influencia
Título	Influencia		
Descripción	El agente debe ser capaz de generar un mapa de influencia para la raza *Terran*.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Media

Tabla 30 Requisito RF-06-01 Influencia

Código	RF-06-02	**Fuente**	CU-06 Atacar
Título	Ataque		
Descripción	El agente debe ser capaz, para la raza *Terran*, de ordenar atacar a las unidades enemigas.		
Necesidad	Esencial	**Prioridad**	Media
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 31 Requisito RF-06-02 Ataque

Código	RF-06-03	**Fuente**	CU-06 Atacar CU-08 Influencia
Título	Ataque + Influencias		
Descripción	El agente basa las órdenes de ataque en base al mapa de influencia para la raza *Terran*.		
Necesidad	Deseable	**Prioridad**	Media
Estabilidad	Estable	**Verificabilidad**	Media

Tabla 32 Requisito RF-06-03 Ataque + Influencias

Código	RF-06-04	**Fuente**	CU-06 Atacar CU-07 Defender
Título	Retirada		
Descripción	Con el objetivo de no perder demasiadas tropas, si resultan muy dañadas, se retiran a la base para la raza *Terran*.		
Necesidad	Deseable	**Prioridad**	Media
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 33 Requisito RF-06-04 Retirada

Código	RF-06-05	**Fuente**	CU-06 Atacar
Título	Agrupar		
Descripción	Las unidades militares deben moverse en grupo y no ir muy separadas para la raza *Terran*.		
Necesidad	Deseable	**Prioridad**	Baja
Estabilidad	Inestable	**Verificabilidad**	Alta

Tabla 34 Requisito RF-06-05 Agrupar

Código	RF-07-01	**Fuente**	CU-07 Defender
Título	Defensa de base		
Descripción	El agente debe ser capaz de situar correctamente a las unidades militares para defender la base para la raza *Terran*.		
Necesidad	Esencial	**Prioridad**	Media
Estabilidad	Estable	**Verificabilidad**	Media

Tabla 35 Requisito RF-07-01 Defensa de base

Código	RF-07-02	**Fuente**	CU-07 Defender
Título	Usar búnkeres		
Descripción	El agente debe ser capaz de llenar los búnkeres con marines para la raza *Terran*.		
Necesidad	Deseable	**Prioridad**	Media
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 36 Requisito RF-07-02 Búnkeres 2

3.4. Diseño del sistema

En este apartado se realiza una evaluación sobre las distintas posibilidades para realizar este agente justificando la opción final escogida. Finalmente se describe en detalle el agente desarrollado siguiendo un orden *top-bottom*, es decir, desde una descripción más global a una más específica de cada componente que lo conforma.

3.4.1. Arquitectura del sistema

El diseño global del mismo se muestra en la Ilustración 29, especificando la relación existente entre cada elemento:

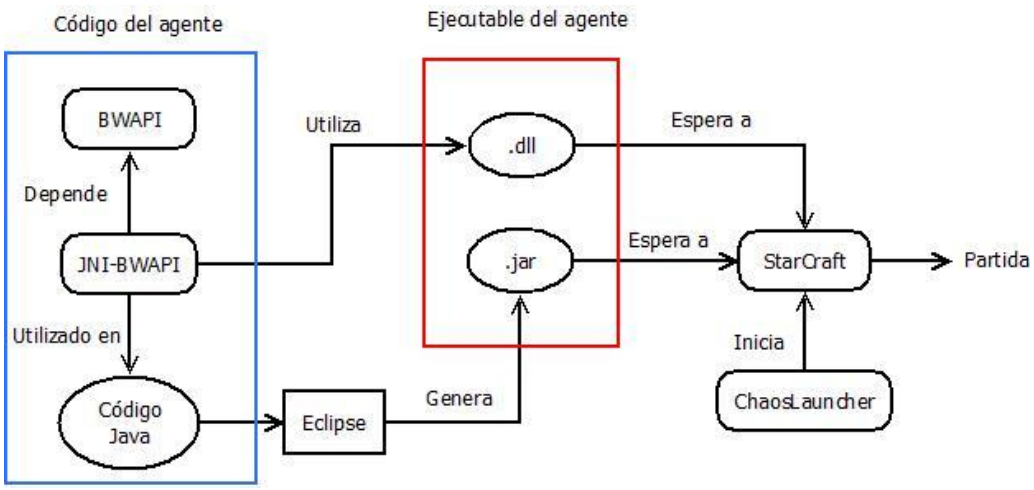


Ilustración 29 Diseño global de la arquitectura del sistema

La ejecución del sistema de forma global (Ilustración 29) sería la siguiente:

1. Código del agente (cuadro azul):
   - Entradas: No recibe.
   - Salidas: código java del agente y librerías .dll de JNI-BWAPI.
   - Funcionamiento: El código java utiliza la API de JNI-BWAPI para su funcionamiento. A su vez, para que JNI-BWAPI pueda funcionar, debe tenerse instalado BWAPI.
2. El código java generado se compilar a través de Eclipse para generar .jar ejecutable que cumpla las dependencias.
3. Ejecutable del agente (cuadro rojo):
   - Entradas: .dll de JNI-BWAPI y el ejecutable .jar generado por Eclipse.
   - Salidas: No se genera ninguna salida.
   - Funcionamiento: El agente se ejecuta a través del .jar generado. Este .jar depende de las librerías .dll de JNI-BWAPI para poder conectarse a StarCraft. Mientras no se inicie ninguna partida en StarCraft, el .jar se encontrará esperando.
4. *ChaosLauncher*: ChaosLauncher, inicia una instancia de StarCraft, para poder conectar el agente y StarCraft, debe ejecutar con el *plugin* del inyector de código.
5. Una vez iniciado StarCraft, se selecciona para jugar una partida y como raza del jugador se selecciona la raza *Terran*.
6. Una vez iniciada la partida, a lo largo de la misma el agente decide que acción realizar con la información que tiene en cada instante de la partida.
7. Se finaliza con la victoria o derrota del agente.

El orden de los pasos 3 y 4 es indiferente, pero deben realizarse antes del paso 5. El paso 6 se repite hasta que finaliza la partida, y se ejecuta en cada *frame* interno del juego. Ejecutarse en cada *frame* del juego implica que los árboles de comportamiento se ejecutan varias veces por segundo, por lo que su carga de trabajo no debe ser muy elevada ya que si no, se bloquea el funcionamiento del juego.

Debido al uso de JNI-BWAPI, la estructura interna de la comunicación entre el agente y *StarCraft* sigue el siguiente esquema (Ilustración 30):

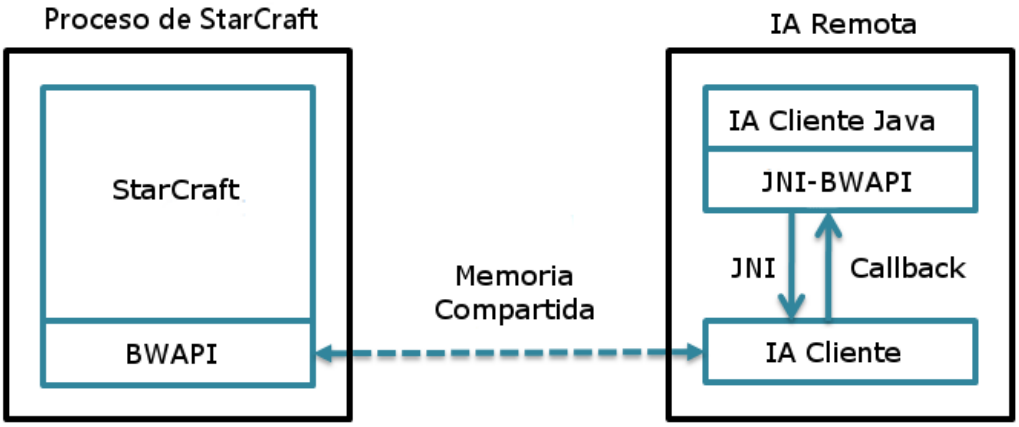


Ilustración 30 Arquitectura general del sistema (Fuente

Esta configuración puede dividirse en dos partes: accesible y no accesible al usuario.

- Accesible al usuario: Está formado por la IA Cliente Java, que representa el código que se puede modificar, concretamente el agente desarrollado que utiliza JNI-BWAPI como API.
- No Accesible al usuario: Está formado por todo lo demás, *StarCraft*, BWAPI, IA Cliente y JNI-BWAPI, consiste en todo aquello que no se puede, o no debe, modificar del código y que depende de terceros.

En la Ilustración 30 se puede apreciar más en detalle la estructura interna y funcionamiento del Ejecutable del agente, de la Ilustración 29. IA Cliente Java y JNI-BWAPI se corresponden con el .jar generado previamente. A través de llamadas de la *Java Native Interface* a la IA simulada por JNI-BWAPI se comunica con BWAPI, que está conectada a StarCraft. Al compartir memoria con BWAPI recibe directamente los resultados y genera respuestas que comunica a JNI-BWAPI, esta comunicación es gracias a las librerías .dll. La comunicación entre JNI-BWAPI y BWAPI (memoria compartida) sólo es posible una vez se inicializa StarCraft a través de *ChaosLauncher* con el inyector de código activado. Todo este proceso de comunicación se realiza en cada *frame* del juego.

3.4.1.1. Descripción general del sistema

En este apartado se detalla a nivel de código la composición del agente desarrollado, que se corresponde con la sección de código accesible al usuario.

La forma más genérica de representar los elementos que forman el agente se muestra en la Ilustración 31:

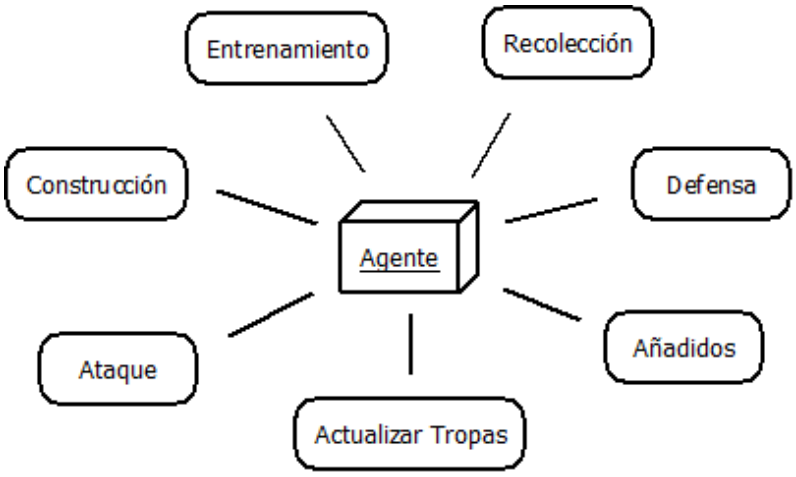


Ilustración 31 Visión global del agente

Todos estos componentes se definirán más en detalle en el apartado 3.4.1.2. Descripción de componentes. A continuación se realiza una breve descripción de la composición y función de cada uno de estos elementos:

- Agente: Se corresponde con el núcleo del sistema y se encarga de gestionar y decidir qué acción realizar en cada instante. En él se implementan los eventos escucha de JNI-BWAPI que permitirán realizar posteriormente acciones en el juego. Es un árbol de comportamiento que actúa como nodo raíz al cual conectar el resto de árboles que se encargan de cada aspecto concreto del juego.
- Recolección: Ilustración 34. Este árbol de comportamiento engloba aquellas acciones relacionadas con la gestión de trabajadores (VCEs). Está formado por tres secuencias: *Reparación de edificios*, *Selección de base* y *Recolección de recursos*.
- Entrenamiento: Ilustración 38. Este árbol de comportamiento se encarga del entrenamiento de nuevas unidades.
- Construcción: Ilustración 40. Este árbol de comportamiento se encarga de la construcción de edificios y mejora de unidades, está formado por dos secuencias distintas: *Construcción de edificios* y *Mejora de unidades.*
- Añadidos: Ilustración 43. Este árbol de comportamiento se encarga de la construcción de los añadidos a edificios.
- Ataque: Ilustración 45. Este árbol de comportamiento se encarga de toda la gestión de ataque de unidades militares.

- Actualizar Tropas: Ilustración 47. Este árbol de comportamiento se encarga de crear, actualizar y mantener las tropas de unidades militares.
- Defensa: Ilustración 50. Este árbol de comportamiento de todo lo relacionado con la defensa de la base. Está formado por dos secuencias: *Bunker* y *Defensa*.

En cada ejecución de la función *matchFrame* de JNI-BWAPI (equivalente a *onFrame* de BWAPI) se ejecutan cada uno de estos árboles, generando las acciones correspondientes dentro del juego.

3.4.1.2. Descripción de componentes

En este apartado se describe mucho más en profundidad todos los elementos indicados en 3.4.1.1. Descripción general del sistema. Se especifica en cada uno los elementos que lo conforman y su funcionamiento.

Agente

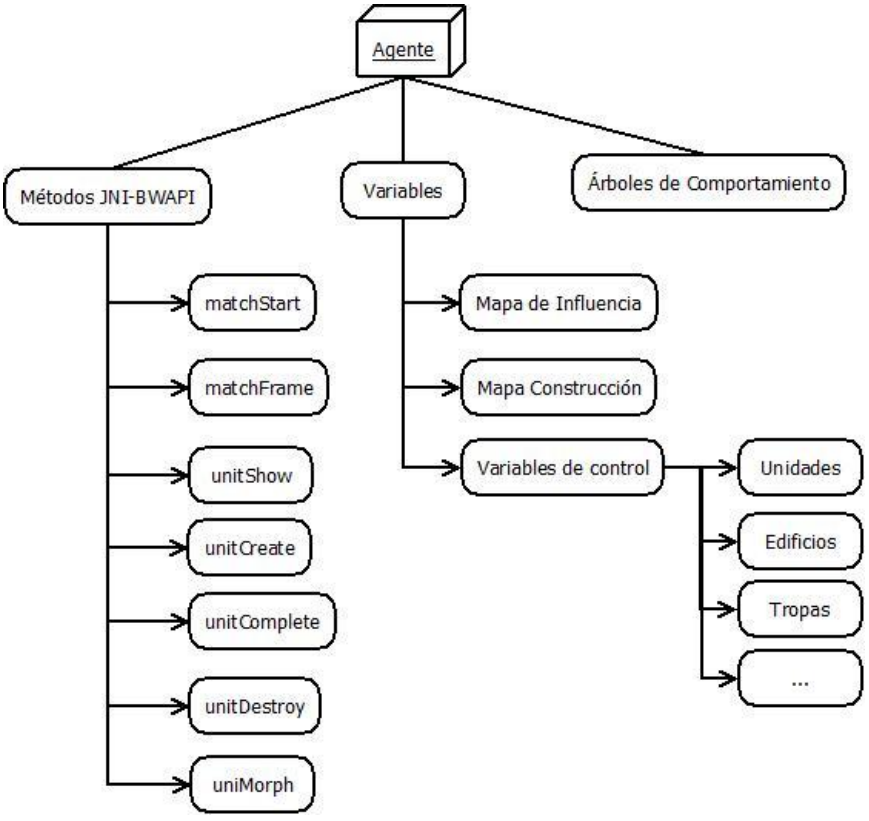


Ilustración 32 Composición del Agente

La Ilustración 32 muestra de forma gráfica la estructura del agente. El agente se compone principalmente de Árboles de Comportamiento y Variables. Mediante estos elementos y los *listeners* de JNI-BWAPI se construye todo el funcionamiento del agente.

Las Variables son el nexo de comunicación entre los *listeners* y los Árboles de Comportamiento. A través de ellas el agente actualiza en cada *frame* la información

que posee en cada momento de la situación del juego a través de *listeners* de JNI-BWAPI.

- **Métodos de JNI-BWAPI:**
    1. matchStart (equivalente a *onStart* de BWAPI): se crean todas las instancias relativas al agente. Se crea la instancia de JNI-BWAPI que se usará durante toda la partida, el mapa de construcción y de influencias inicial y se les da valores a variables como el número de suministros o listas de entrenamiento. Una vez creadas las instancias del agente se definen y crean las estructuras de los árboles de comportamiento. La implementación de los nodos hoja se realiza en clases separadas.
    2. matchFrame (equivalente a *onFrame* de BWAPI): el primer paso en este método es ejecutar los árboles de comportamiento definidos en matchStart. Su orden de ejecución es: *Recolección*, *Construcción*, *Añadidos*, *Entrenamiento*, *Defensa*, *Actualización de tropas* y finalmente *Ataque*. El mapa de influencias se actualiza si el número de *frames* transcurridos es múltiplo de 300.
    3. unitShow (equivalente a *onUnitShow* de BWAPI): se utiliza para identificar la raza del enemigo y establecer las unidades que se deben entrenar en la partida.
    4. unitCreate (equivalente a *onUnitCreate* de BWAPI): cada vez que comienza la creación de un edificio del agente, se añade a la lista de edificios pendientes.
    5. unitComplete (equivalente a *onUnitComplete* de BWAPI): cada vez que se finaliza el entrenamiento de una unidad o construcción de un edificio se actualiza el Mapa de influencias para contemplar esta nueva unidad. Si la unidad es del agente, se actualizan las variables de control correspondientes y el Mapa de construcción.
    6. unitDestroy (equivalente a *onUnitDestroy* de BWAPI): en este método se actualizan las listas de control correspondientes, eliminándola de donde corresponda, para así evitar problemas de actualización o influencias.
    7. *unitMorph* (equivalente a *onUnitMorph* de BWAPI): debido a cuestiones internas de la API, se utiliza para conocer cuando se finaliza la construcción de la refinería.
- **Variables:**
    1. Mapa de influencia: consiste en una representación matricial del mapa de juego. Cada casilla tiene asignado un valor numérico (influencia): valores positivos si es del jugador, valores negativos si es de los enemigos y 0 si es neutral. Un mayor valor indica un mayor control por parte del jugador/enemigo de esa posición. La influencia es generada por las unidades, ya sean edificios o militares, y actualizada cada 300 iteraciones de juego.

2. Mapa de Construcción: matriz de dimensiones X x Y que se corresponden con el ancho y alto del mapa actual. Cada casilla posee un valor entre 0 y 6, indica el tamaño máximo del edificio que se puede construir en esa posición si tomamos la esquina superior izquierda del edificio como referencia.
3. Variables de control: conjunto de listas y variables que sirven principalmente para mantener un control propio sobre el transcurso del juego. También se utilizan para ahorrar tiempos de cómputo al tener de forma directa acceso a unidades o edificios concretos.

- **Árboles de Comportamiento:** el orden de ejecución y motivo de los que árboles de comportamiento se puede observar en la Ilustración 33, siguiendo un orden de arriba a abajo y de izquierda a derecha:

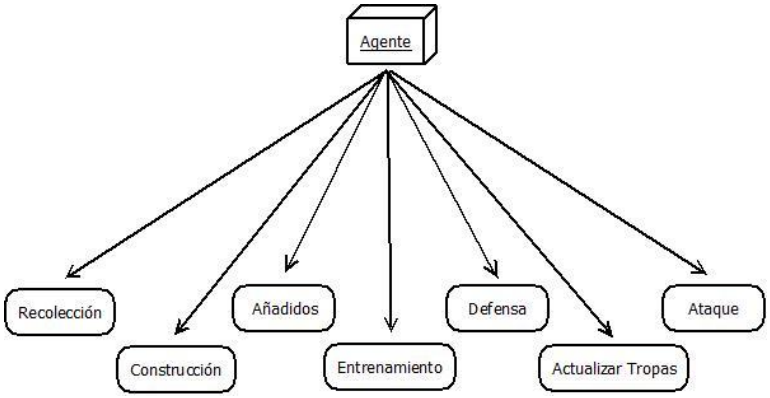


Ilustración 33 Visión del agente en árbol

1. Recolección: se ejecuta el primero debido a la importancia que tiene la recolección en el juego. Si no se recolectan recursos no se pueden realizar correctamente el resto de comportamientos.
2. Construcción: después de recolectar recursos, lo segundo más importante es construir edificios, ya que desde los edificios se van a crear nuevas unidades con la que defender y atacar.
3. Añadidos: el orden de ejecución este árbol es indiferente. Se encarga de construir añadidos a los edificios previamente construidos.
4. Entrenamiento: Se realiza después de construcción para dar posibilidad a que existan edificios donde entrenar unidades, y antes de cualquier acción de ataque o defensa, ya que sin unidades no tiene sentido atacar o defender.
5. Defensa: Se realiza antes de cualquier acción de ataque porque es importante defender correctamente la base antes de atacar al enemigo.
6. Actualizar Tropas: Se realiza antes del ataque para así poder optimizar las listas de control de tropas (juntando tropas existentes o eliminando tropas vacías).
7. Ataque: Se realiza después de actualizar las tropas para asegurar que las variables de control de las tropas están correctas.

Mapa de influencia

El mapa de influencia es uno de los elementos presentes entre las variables existentes en el Agente. Es fundamental para los movimientos de ataque y defensa de las unidades.

Un mapa de influencia consiste en una representación del mapa de juego en casillas. Cada casilla tiene asociado una influencia que representa el control que posee el jugador o los enemigos de la casilla.

La influencia se representa con valores numéricos, siendo los valores positivos los asociados al jugador y sus aliados y valores negativos los asociados para el enemigo. Un valor 0 indica una casilla neutral. Esta influencia es generada por las unidades y edificios presentes en el juego, cada una con valores específicos. Se debe actualizar a medida que avanza la partida, ya que con su movimiento, creación y pérdida de unidades las zonas de control varían.

El mapa de influencia es usado para determinar qué zonas atacar, porque las casillas con valores negativos serán lugares donde el enemigo tiene unidades o edificios. Está representado internamente como una matriz de dimensiones X x Y, donde son el ancho y alto del mapa respectivamente.

Recolección

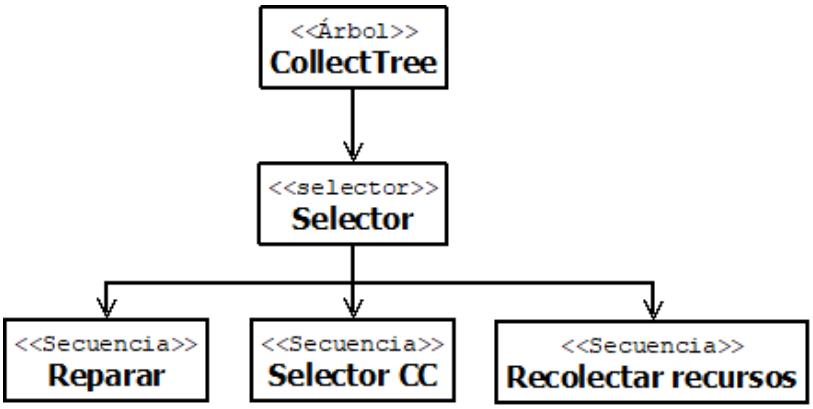


Ilustración 34 Árbol de Comportamiento: Recolección

Este árbol de comportamiento (Ilustración 34) engloba aquellas acciones relacionadas con la gestión de VCEs. Está formado por tres secuencias que se explican más en detalle a continuación: *Reparar*, *Seleccionar CC* y *Recolección de recursos*.

El orden de las secuencias se debe a las siguientes justificaciones:

- Reparar la primera, ya que si existe algún edificio dañado, no hay que perder tiempo y repararlo.
- Selector CC sirve para elegir sobre qué *Command Center* se van a centrar las acciones, y dado que los recursos estás asociados a un centro de mando, no

se puede ordenar a los trabajadores que recolecten recursos sin saber antes los recursos disponibles.

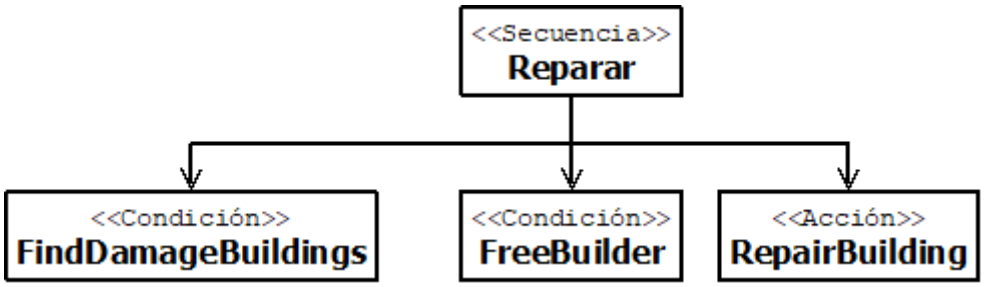


Ilustración 35 Secuencia: Reparar

La secuencia *Reparar* (Ilustración 35) se encarga de encontrar edificios dañados y ordenar a un trabajador libre que lo repare. Está formada por los siguientes componentes:

- FindDamageBuildings (Condición): encuentra y guarda los edificios dañados actualmente en una lista. Si encuentra edificios dañados devuelve verdadero, en otro caso falso.
- FreeBuilder (Condición): encuentra un trabajador libre para reparar uno de los edificios dañados. Si hay trabajadores libres devuelve verdadero, en caso contrario falso.
- RepairBuilding (Acción): ordena al trabajador libre que repare el primer edificio de la lista de dañados. Devuelve el resultado de la orden de reparación.

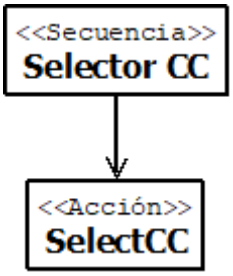


Ilustración 36 Secuencia: Seleccionar CC

La secuencia *Seleccionar CC* (Ilustración 36) se encarga de escoger un *Command Center* de todos los construidos. Está formada por una única acción:

- SelectCC (Acción): selecciona de todos los Centros de Mandos construidos, aquel que no tenga entrenados al menos 20 trabajadores. En otros casos, selecciona uno aleatoriamente.

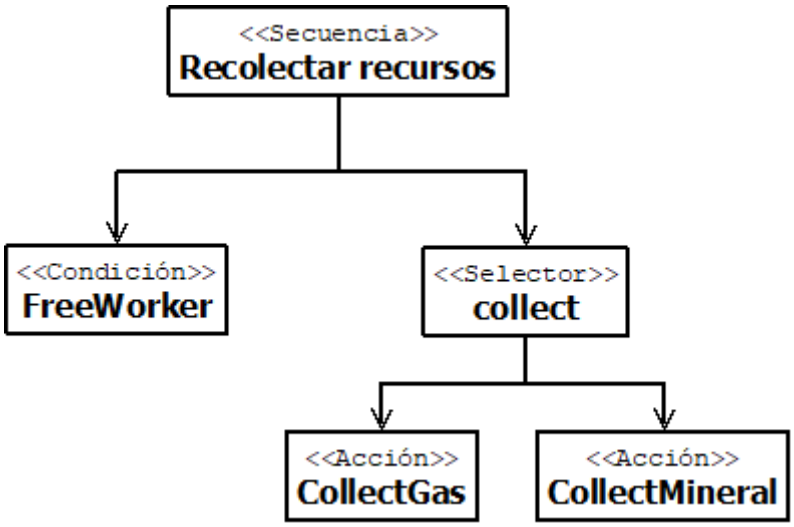


Ilustración 37 Secuencia: Recolectar

La secuencia *Recolectar recursos* (Ilustración 37) se encarga de poner a los trabajadores libres a recolectar recursos. Está formada por los siguientes componentes:

- FreeWorker (Acción): comprueba que existan trabajadores inactivos, para ello comprueba la lista de control que contiene a todos los trabajadores. De ser así devuelve verdadero, en otro caso falso.
- Collect (Selector): intenta primero asignar el trabajador a recolectar gas vespeno y si no es posible, lo pone a recolectar minerales. El número de trabajadores óptimo para recolectar eficientemente un recurso es tres porque mientras uno está recolectando, otro está entregando en la base el recurso y otro se encuentra esperando para recolectar inmediatamente cuando finalice el trabajador que está actualmente recolectando.
    - CollectGas (Acción): si existe una refinería y no hay tres trabajadores actualmente asignados, le asigna a gas vespeno y devuelve verdadero, en cualquier otro caso, devuelve falso para así pasar a recolectar minerales.
    - CollectMineral (Acción): si el número de VCEs asignados a mineral no llega a 3 VCE por nodo de mineral, se le asigna a recolectar minerales.

Entrenamiento

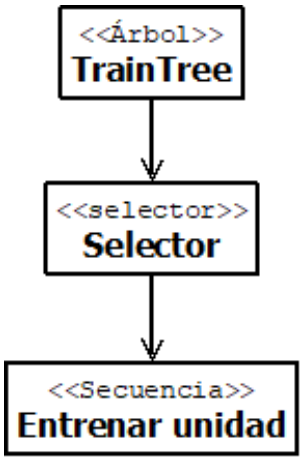


Ilustración 38 Árbol de Comportamiento: Entrenamiento

Este árbol de comportamiento (Ilustración 38) se encarga del entrenamiento de nuevas unidades. Está formado por una única secuencia que se explica más en detalle a continuación. Al tratarse de una única secuencia el selector no es necesario, pero se encuentra para poder añadir más comportamientos en el futuro sin tener que modificar el árbol en profundidad.

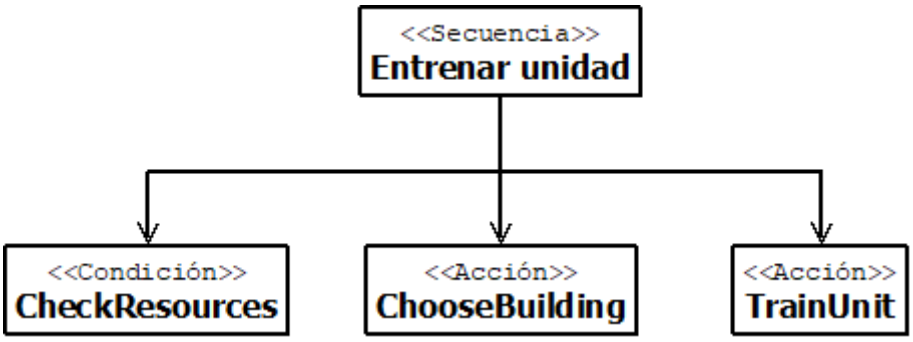


Ilustración 39 Secuencia: Entrenar

La secuencia *Entrenar unidad* (Ilustración 39) se encarga de comprobar que es posible entrenar una unidad concreta y de ser posible, ponerla a entrenar. Está formada por los siguientes componentes:

- CheckResources (Condición): comprueba que se posean los recursos suficientes (mineral y gas vespeno) para pagar el coste de entrenar la unidad. Si se poseen los recursos devuelve verdadero, falso en caso contrario.
- ChooseBuilding (Acción): encuentra el edificio correspondiente para entrenar la unidad.

- TrainUnit (Acción): acción que pone a entrenar la unidad en el edificio anteriormente encontrado. Adicionalmente, esta acción tiene una serie de decoradores para garantizar que se entrenan unidades de forma proporcionada, por ejemplo:
    - Un máximo de tres VCE por refinería de gas vespeno, ya que es el número eficiente de trabajadores para recolectarlo eficientemente.
    - Un médico por cada cinco marines + murciélagos de fuego, para mantener un el número de unidades de cada tipo equilibrado y que haya suficientes médicos para curar cuando las unidades sean atacadas, pero no demasiados médicos como para que falte daño (ya que los médicos no atacan).

Construcción

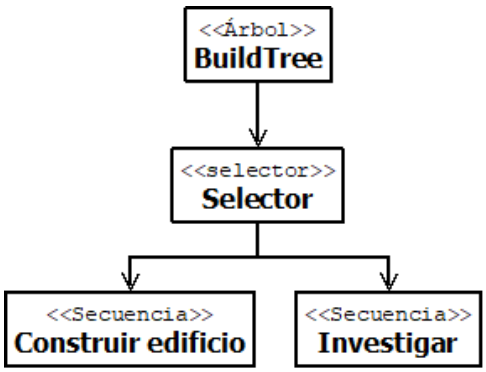


Ilustración 40 Árbol de Comportamiento: Construcción

Este árbol de comportamiento (Ilustración 40) se encarga de la construcción de edificios y mejora de unidades. Está formado por dos secuencias distintas que se explican más en detalle a continuación: *Construcción de edificios* y *Mejora de unidades*.

El orden de estas secuencias se debe a que la investigación de mejoras es un proceso costoso y secundario ya que a comienzos de la partida es más importante construir los edificios necesarios para poder avanzar que investigar mejoras.

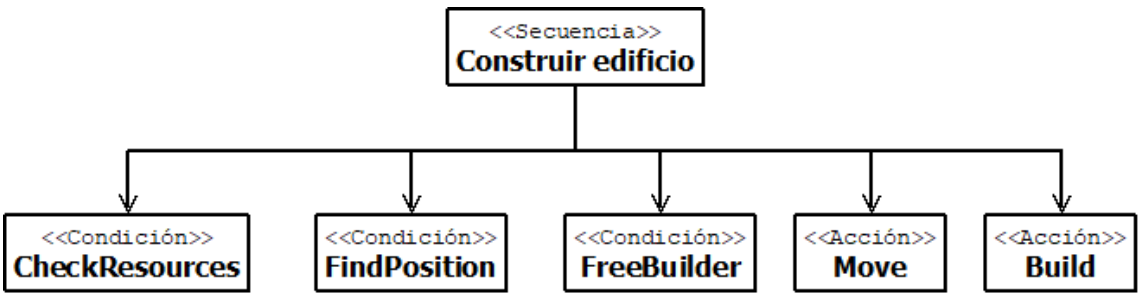


Ilustración 41 Secuencia: Construir

La secuencia *Construir edificio* (Ilustración 41) se encarga de comprobar recursos, encontrar una posición y trabajador libre y ordenarle construir. Está formada por los siguientes componentes:

- CheckResources (Condición): comprueba que se posean los recursos suficientes (mineral y gas vespeno) para pagar el coste de construir el edificio. Si se tienen los recursos devuelve verdadero, falso en caso contrario.
- FindPosition (Condición): se encarga de verificar que existe una posición válida para construir el edificio correspondiente. Dependiendo del mapa, del número de *choke points* cercanos a la base y del edificio a construir existen dos formas de buscar una posición:
    - Lejos del *choke point*: cuando sólo existe una entrada a la base. Se usa para edificios como los barracones o la fábrica, al ser edificios importantes deben situarse lejos de la entrada para estar más protegidos.
    - Cerca de la base: cuando existen varios puntos de entrada a la base. En este caso, los barracones o fábricas se construyen cerca de la base y el resto de edificios más lejos, de forma que los más importantes se encuentren protegidos.

  Si encuentra una posición válida devuelve verdadero, en caso contrario falso. Adicionalmente, para ahorrar tiempos de cómputo, se tiene una serie de decoradores para controlar la construcción de edificios y maximizar los resultados, principalmente órdenes de construcción y límite de número de edificios de cierto tipo.
- FreeBuilder (Condición): encuentra un trabajador libre para construir el edificio deseado. Si hay trabajadores libres devuelve verdadero, en caso contrario falso.
- Move (Acción): para poder construir no debe existir Niebla de Guerra en esa posición, por lo tanto, antes de construir se mueve el trabajador hacia la posición concreta. Devuelve verdadero si se puede mover, falso en otro caso.
- Build (Acción): ordena al trabajador seleccionado que construya el edificio en cuestión. Devuelve verdadero si la orden se está ejecutando con éxito, falso en otro caso.

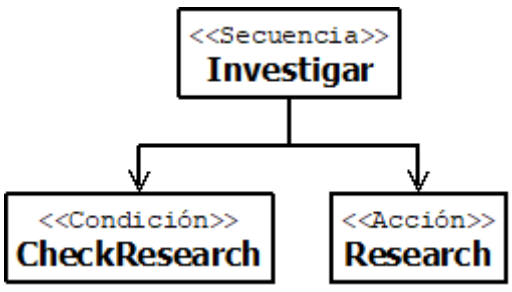


Ilustración 42 Secuencia: Investigar

La secuencia *Investigar* (Ilustración 42) se encarga en caso de que no sea necesario construir nuevos edificios, de investigar tecnologías para mejorar a las unidades. Está formada por los siguientes componentes:

- CheckResearch (Condición): comprueba que se posean los recursos suficientes (mineral y gas vespeno) para pagar el coste de realizar la investigación y que existe el edificio correspondiente entre todos los construidos. Devuelve verdadero al cumplirse estas condiciones, falso en cualquier otro caso.
- Research (Acción): selecciona el edificio en concreto y comienza a realizar la investigación. Si todo funciona correctamente devuelve verdadero, falso cualquier otro caso.

Añadidos

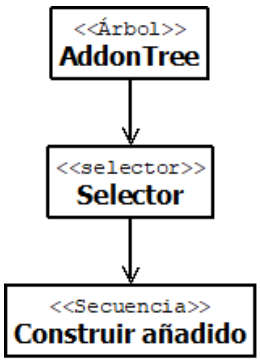


Ilustración 43 Árbol de Comportamiento: Añadidos

Este árbol de comportamiento (Ilustración 43) se encarga de la construcción de los añadidos a edificios. Está formado por una única secuencia que se explica más en detalle a continuación. Al tratarse de una única secuencia el selector no es necesario, pero se encuentra para poder añadir más comportamientos en el futuro sin tener que modificar el árbol en profundidad.

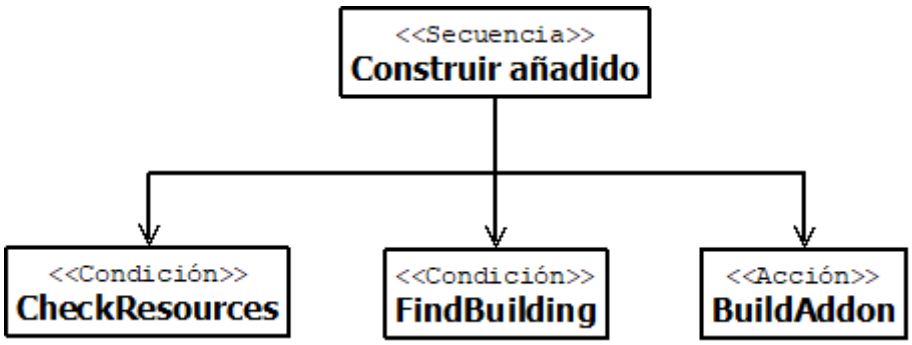


Ilustración 44 Secuencia: Añadidos

La secuencia *Construir añadido* (Ilustración 44) se encarga de comprobar recursos, encontrar el edificio correspondiente y construir el añadido. Está formada por los siguientes componentes:

- CheckResources (Condición): comprueba que se posean los recursos suficientes (mineral y gas vespeno) para pagar el coste de la ampliación. Devuelve verdadero si se cumple la condición, falso en otro caso.

- FindBuilding (Condición): comprueba que existe y está completado el edificio del que se quiere construir la ampliación y que no tiene construida la misma. Devuelve verdadero si existe el edificio y no tiene ampliación, falso en cualquier otro caso.
- BuildAddon (Acción): selecciona el edificio en concreto y ordena construir el añadido del mismo. Si puede construir el añadido devuelve verdadero, falso en cualquier otro caso.

Ataque

Para poder gestionar fácilmente tanto el ataque como la defensa se ha implementado una clase *Troop*. Esta clase contiene los siguientes atributos:

- Unidades: lista que contiene todas las unidades asignadas a la tropa.
- Destino: posición destino a la que se van a mover las unidades.
- Estado: valor numérico. Estado actual de la tropa, hay cinco estados posibles:
  - 0 – Nada: La tropa no está haciendo nada.
  - 1 – Atacar: Se le ha ordenado que ataque al enemigo.
  - 2 – Defender: Se le ha ordenado defender cierta posición.
  - 3 – Reagrupar: Se están reagrupando las unidades.
  - 4 – Retirada: La tropa ha perdido muchas unidades y se retira a la base.
  - 5 – Esperando: La tropa ha finalizado la acción que estaba realizando previamente y se encuentra esperando.
- Detector: valor booleano que indica si la tropa tiene en su lista de unidades un detector para poder detectar unidades invisibles.

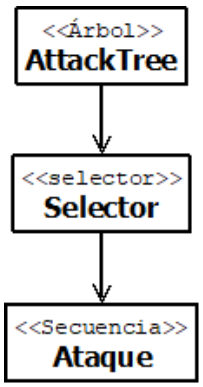


Ilustración 45 Árbol de Comportamiento: Ataque

Este árbol de comportamiento (Ilustración 45) se encarga de toda la gestión de ataque de unidades militares. Está formado por una única secuencia que se explica más en detalle a continuación. Al tratarse de una única secuencia el selector no es necesario, pero se encuentra para poder añadir más comportamientos en el futuro sin tener que modificar el árbol en profundidad.

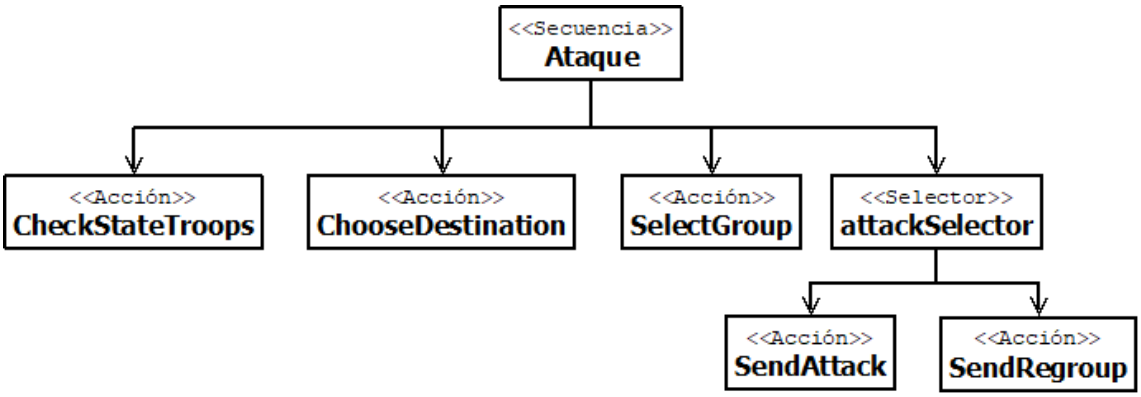


Ilustración 46 Secuencia: Ataque

La secuencia *Ataque* (Ilustración 46) se encarga de comprobar el estado de las tropas existentes, encontrar una posición para atacar, escoger una tropa y dependiendo de la tropa escogida, ordenarle atacar o reagruparse. Está formada por los siguientes componentes:

- CheckStateTroops (Acción): comprueba el estado actual de las tropas, elimina de la lista de control de tropas aquellas con cero unidades, comprueba si tiene alguna unidad capaz de revelar unidades invisibles (unidad detectora) entre las unidades y si la tropa ha perdido muchas unidades le ordena retirarse.
- ChooseDestination (Acción): obtiene la posición de ataque para las tropas. Se basa en la influencia y cercanía a las bases, dando más prioridad a la cercanía de ellas.
- SelectGroup (Acción): selecciona una de las tropas existentes para la acción a realizar. Se seleccionan sólo tropas con estado 0, 5 o 1, en ese orden concretamente. No se seleccionan tropas con estado 2, por que deben quedarse defendiendo la posición, ni estado 3 ya que se están reagrupando, ni estado 4, ya que si se están retirando no pueden luchar.
- AttackSelector (Selector): este selector elige si ordenar atacar o reagruparse a las unidades.
    - SendAttack (Acción): primero comprueba si las unidades se encuentran cerca unas de otras, de no ser así devuelve falso. Si las unidades se encuentran agrupadas, se les ordena atacar la posición objetivo y se devuelve verdadero.
    - SendRegroup (Acción): en caso de estar las unidades muy alejadas unas de otras, se les ordena agruparse, moviéndose todas a una posición cercana. Siempre devolverá verdadero.

Actualizar Tropas

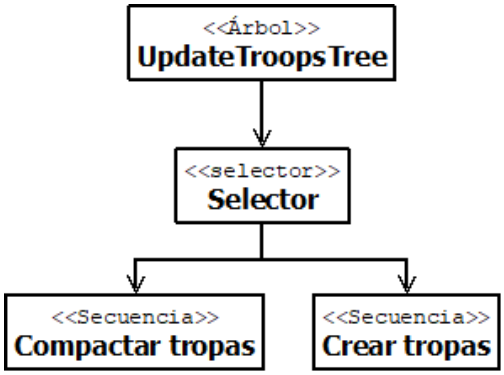


Ilustración 47 Árbol de Comportamiento: Actualizar Tropas

Este árbol de comportamiento (Ilustración 47) se encarga de crear, actualizar y mantener las tropas de unidades militares. Está formado por dos secuencias que se explican más en detalle a continuación: *Compactar tropas* y *Crear tropas*.

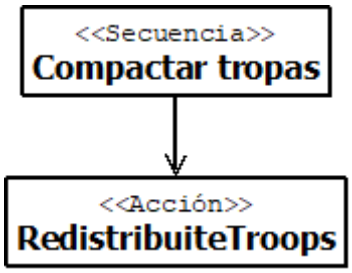


Ilustración 48 Secuencia: Compactar Tropas

La secuencia *Compactar tropas* (Ilustración 48) se encarga de redistribuir las unidades entre las tropas existentes. Está formada por una única acción:

- RedistribuiteTroops (Acción): Si una tropa tiene menos de diez unidades y se encuentra esperando (estado 5), retirándose (estado 4) o sin hacer nada (estado 0), las unidades pertenecientes a esa tropa van a pasar a formar parte de otra tropa que tenga también menos de diez unidades y tenga estado 4 o 5.

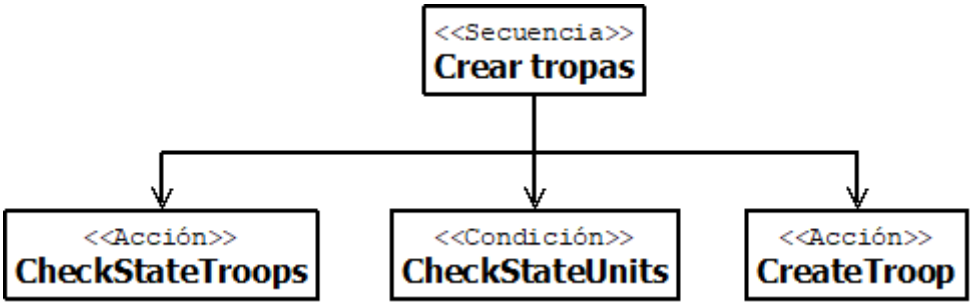


Ilustración 49 Secuencia: Crear Tropas

La secuencia *Crear tropas* (Ilustración 49) se encarga de generar nuevas tropas cuando todas las existentes están llenas o no existe ninguna. Está formada por los siguientes componentes:

- CheckStateTroops (Acción): comprueba el estado actual de las tropas, elimina tropas con cero unidades, comprueba si tiene alguna unidad detectora entre las unidades y si la tropa ha perdido muchas unidades le ordena retirarse.
- CheckStateUnits (Condición): comprueba que existen suficientes unidades sin asignar para crear una nueva tropa.
- CreateTroop (Acción): asigna todas las unidades sin asignar a las tropas que tengan menos de 10 unidades. Si todas las tropas se encuentran llenas, crea una tropa vacía y la añade a la lista de control de tropas existentes, para posteriormente asignarlas a esa nueva tropa.

Defensa

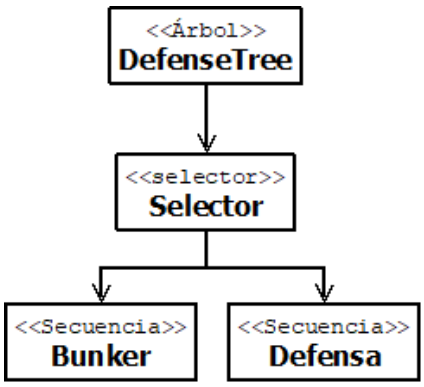


Ilustración 50 Árbol de Comportamiento: Defensa

Este árbol de comportamiento (Ilustración 50) de todo lo relacionado con la defensa de la base. Está formado por dos secuencias que se explican más en detalle a continuación: *Bunker* y *Defensa*.

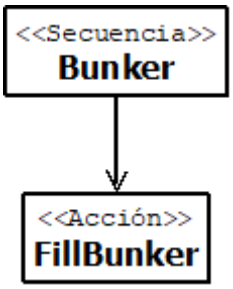


Ilustración 51 Secuencia: Búnker

La secuencia *Bunker* (Ilustración 51) se encarga de rellenar los búnkeres existentes con marines. Está formada por una única acción:

- FillBunker (Acción): selecciona un marine sin asignar y le ordena entrar en uno de los búnkeres libres existentes. Si existen búnkeres libres y es capaz de ordenar a un marine que entre devuelve verdadero, si no hay ningún marine libre o todos los búnkeres existentes están llenos devuelve falso.

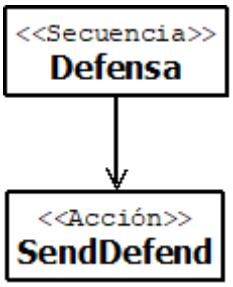


Ilustración 52 Secuencia: Defensa

La secuencia *Defensa* (Ilustración 52) se encarga de ordenar a las unidades militares inactivas que se muevan a la posición defensiva correspondiente. La posición defensiva se calcula al comienzo de la partida dependiendo del número de *choke points* de entrada a la base:

- Si existe un único *choke point* de entrada a la base, se considera posición defensiva esa zona.
- Si existen varias entradas se toma como posición defensiva, la zona alrededor del *Command Center*.

Está formada por una única acción:

- SendDefend (Acción): ordena a las unidades recién creadas (no tienen ninguna acción asignada) que se muevan a la considerada posición defensiva. Si el número de unidades en esa posición es mayor que 10 devuelve falso, en caso contrario, verdadero.

Descripción del funcionamiento del sistema

En este apartado se muestra un diagrama (Ilustración 53) sobre la ejecución del agente a lo largo de una partida.

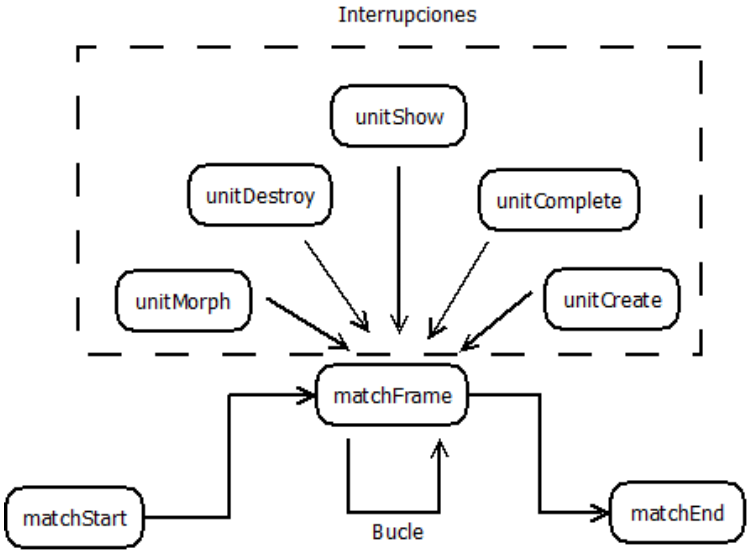


Ilustración 53 Bucle de juego

A lo largo de una partida se activan principalmente tres funciones de JNI-BWAPI:

- **matchStart**: se ejecuta una sola vez al comenzar la partida. En esta función se deben inicializar datos, estructuras, variables, etc., todo lo necesario para que funcione correctamente el agente.
- **matchFrame**: se ejecuta continuamente durante la partida, ya que representa el bucle de juego. En esta función se ejecutan los árboles de comportamiento definidos en 3.4.1.2. Descripción de componentes . Además, se ve interrumpida por otras funciones que se ejecutan cuando ocurren eventos dentro del propio juego, como son:
    - unitMorph: se ejecuta cuando una unidad se transforma en otra. En el agente se utiliza para conocer cuando se construye una refinería.
    - unitDestroy: se ejecuta cuando una unidad es destruida. En el agente se utiliza para mantener actualizadas listas de control, tales como: unidades entrenadas, edificios construidos o tropas existentes. Además, también se utiliza para actualizar el mapa de influencia (eliminar la influencia de la unidad eliminada) y el mapa de construcción en caso de ser un edificio (liberar el espacio ocupado).
    - unitShow: se ejecuta cuando una unidad enemiga sale de la niebla de guerra. En el agente se utiliza para conocer la raza enemiga y definir las unidades a entrenar.
    - unitComplete: se ejecuta cuando el juego considera que una unidad se ha entrenado o un edificio se ha acabado de construir. En el agente se utiliza para mantener actualizadas listas de control, tales como: unidades entrenadas, edificios construidos o tropas existentes. Además, también se utiliza para actualizar el mapa de influencia (añadir la influencia de la unidad o edificio creado) y el

mapa de construcción en caso de ser un edificio (marcar el espacio ocupado).

- unitCreate: se ejecuta cuando en el juego se comienza a entrenar una unidad o construir un edificio (no está finalizado). En el agente se utiliza para mantener actualizadas las listas de edificios en construcción para así no construir varias veces un mismo edificio, ya que se considera que el edificio está construido cuando se llama la función *unitComplete.*

- **matchEnd**: se ejecuta una sola vez cuando finaliza la partida.

Capítulo 4: Experimentación

En este capítulo se muestra y analiza la experimentación realizada durante cada versión del agente, observando la evolución en cada una de ellas.

4.1. Mapas utilizados.

Los mapas utilizados han sido cuatro: *Astral Balance*, *Fire Walker*, *Breaking Point y Full Circle*. Una vista de cada de cada mapa puede verse en la Ilustración 54 Vista de cada mapa.

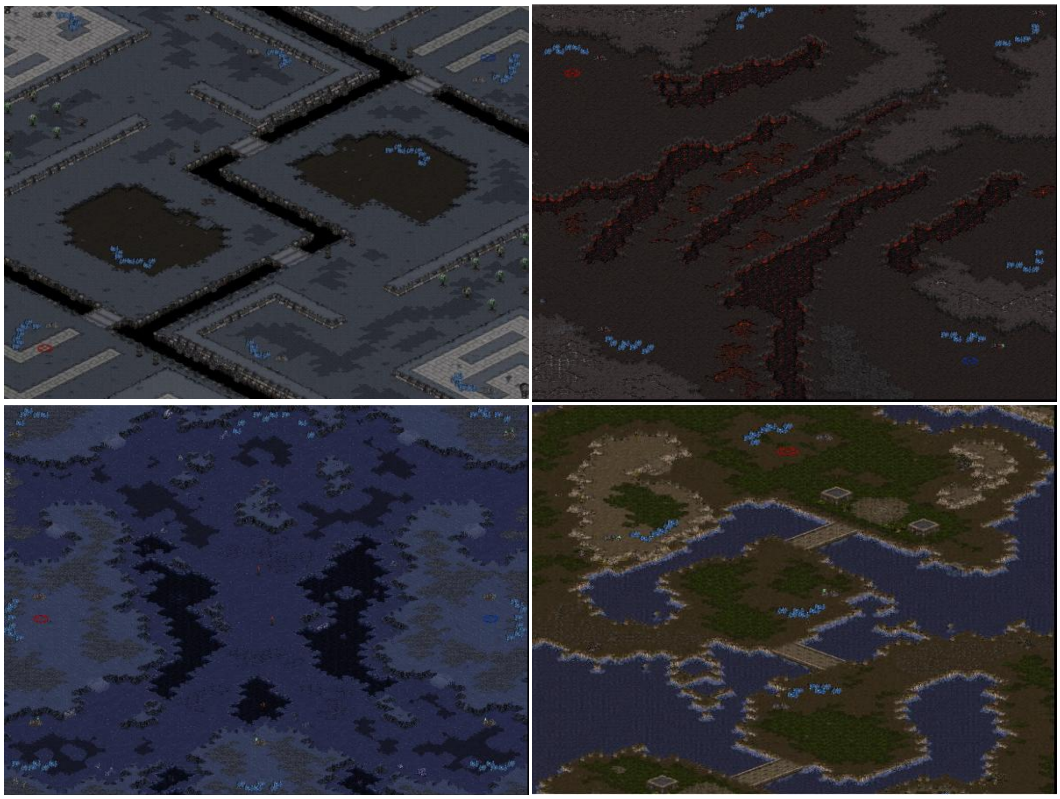


Ilustración 54 Vista de cada mapa

Las decisiones de la elección de cada mapa han sido:

- ***Astral Balance***: Mapa con una única entrada a la base y varias expansiones disponibles que se encuentran en cierto grado protegidas (única entrada y/o distinto nivel de altura). Posee zonas estrechas por las que pasar y varios niveles de altura, incluidos niveles sólo accesibles por vehículos aéreos.
- ***Fire Walker***: Similar a *Astral Balance*, pero sin zonas inaccesibles. Posee una única entrada a la base y expansión cercana.
- ***Breaking Point***: Mapa con múltiples expansiones y varias entradas a la base. La estructura del mapa sólo permite pasar por dos caminos estrechos para atacar al enemigo, que fuerzan a las unidades a pasar varias zonas de expansión.

- ***Full Circle***: Mapa con múltiples expansiones y varias entradas a la base. Todo el mapa es accesible con unidades terrestres. Existe un único camino principal que recorre casi todas las expansiones.

4.2. Evolución de la influencia.

El mapa de influencia es la técnica principal para controlar el ataque y la defensa, aunque también la construcción de ciertos edificios (en menor medida) a lo largo de la partida. Sin él no se podría ganar ni tampoco aguantar los ataques enemigos.

Debido a su importancia y a su uso, a lo largo de las distintas versiones se han ido ajustando los valores de influencia asignados a cada unidad y su actualización.

4.2.1. Influencia en la versión 0

En la versión inicial se aplican los siguientes criterios para aplicar y actualizar el mapa de influencia:

- Se actualiza cada 300 *frames*.
- Se aplican los siguientes valores de influencia si es un edificio:
  - Edificios capaces de atacar: Se aplica una influencia de 4. Abarca los edificios que cumplen la condición "*isAttackCapable*()[1]".
  - Defensivo: Se aplica una influencia de 5. Esta categoría abarca búnkeres (Terran), Torretas de misiles (Terran) y Colonia de esporas (Zerg).
  - Otros: Se aplica una influencia de 3. Esta categoría abarca el resto de edificios.
- Se aplican los siguientes valores de influencia si es una unidad:
  - Mecánica: Se aplica una influencia de 3. Esta categoría abarca a todos los vehículos de movimiento terrestre.
  - Aérea: Se aplica una influencia de 6. Esta categoría abarca todas las naves.
  - Otros: Se aplica una influencia de 1. Esta categoría abarca el resto de unidades.
- La actualización de influencia se aplica en caso de que la unidad se haya desplazado. En ese caso, se aplica la nueva influencia y se elimina la anterior.

El aspecto más problemático de este mapa de influencias es la actualización, que como se puede ver en las pruebas realizadas durante la versión 0 (enlace), deriva en grupos de unidades que no cambian de posición, ya que se encuentran en zonas de influencia negativa y al no desplazarse las tropas, esa influencia no se actualiza y por tanto no desaparece.

[1] bwapi.github.io/class_b_w_a_p_i_1_1_unit_type.html#af1895e9778ff7cefbf8ab6bce0370573

4.2.2. Influencia en la versión 1

En esta versión todos los valores y métodos se mantienen igual que en la versión 0, ya que al tratarse de una versión centrada en la construcción la influencia no afecta.

4.2.3. Influencia en la versión 2

En esta versión se aplican los siguientes cambios en la aplicación de la influencia:

- Los valores de influencia se mantienen igual que en la versión anterior, pero ahora se aplican solamente si la unidad en concreto no es trabajadora, ya que la influencia que aportaban no era relevante para el ataque y defensa y sólo aportaba ruido al cómputo de la influencia.

Este cambio, aunque pequeño, arregla parcialmente un problema que se puede observar en las pruebas de la versión 0. Ahora las tropas se quedarán menos tiempo bloqueadas en los minerales de la base enemiga al haber eliminado la influencia que introducían los trabajadores.

4.2.4. Influencia en la versión 3

En esta versión se han realizado cambios significativos en la aplicación y actualización de la influencia, lo cual elimina casi totalmente los bloqueos en el movimiento.

- Separación de la actualización de la influencia de los edificios y de las unidades, ya que se tratan de forma distinta. Los edificios sólo deben aplicar influencia al mapa cuando se crean y, por tanto, se mantiene constante hasta que son destruidos. Las unidades por otro lado, aplican la influencia cuando se crean, pero como pueden moverse, su influencia debe ser actualizada a lo largo de la partida, eliminándola de su posición anterior y aplicándose a la nueva.
- Nuevas influencias para edificios:
    - Bases: Se aplica una influencia de 7. Esta categoría incluye las bases de cada raza y se les aplica la máxima influencia, ya que son edificios clave.
- Nueva influencia para unidades:
    - Otros: Se pasa de aplicar una influencia de 1 a aplicar una influencia de 2.
    - Resto de influencias de mantienen igual (3 y 6).
- Nueva actualización de influencias: La nueva influencia que se utiliza para actualizar el mapa es la de las unidades, no la de los edificios, y no depende de si las unidades cambian de posición. El proceso para actualizar la influencia consiste en: eliminar la influencia de la anterior posición y aplicar la influencia de la unidad en cuestión en su posición actual.

Esta nueva actualización de la influencia junto con los nuevos valores aplicados a edificios y unidades elimina finalmente el bloqueo del movimiento debido a la influencia, puede observarse claramente en las pruebas de la versión 3 (enlace).

4.2.5. Influencia en la versión 4

Esta versión aplica unos cambios menores de control de errores y nuevas influencias para edificios y unidades:

- Nueva influencia para edificios: Se han aplicado nuevos criterios y orden de aplicación de influencias.
    - Edificios importantes: Se aplica una influencia de 5. Esta categoría abarca edificios como: Pilones (Protoss), Búnkeres (Terran), Colonia de Esporas y Colonia Hundida (Zerg), son edificios a tener en cuenta, ya sea por su importancia (Pilones) o por que indican una zona bien defendida (Búnker y Colonias).
    - Edificios capaces de atacar: Se aplica una influencia de 4. Esta categoría sigue abarcando a todos los edificios que cumplen la condición de método "*isAttackCapable()*".
    - Bases: Se mantienen como hasta ahora, se aplica una influencia de 7 y abarca las bases de cada raza.
    - Otros: Se mantienen como hasta ahora, se aplican una influencia de 3.
- Nuevas influencias para unidades: Todas las influencias se han cambiado, reduciendo sus valores.
    - Mecánicas: Su valor pasa de 3 a 2.
    - Aéreas: Su valor pasa de 6 a 3.
    - Otros: Su valor pasa de 2 a 1.

Los cambios más significativos son la aplicación del orden de la influencia, debido a que en las versiones anteriores se aplicaban de forma errónea, ya que una "Colonia de Esporas" estaba considerada un edificio defensivo (influencia 5). Sin embargo, también puede atacar y al estar primera la evaluación de "*isAttackCapable*()", se le aplicaba una influencia de 4 en vez de 5. Adicionalmente se ha aprovechado y se han incluido dos edificios importantes: el Pilón, ya que es la espina dorsal de las construcciones Protoss y la Colonia Hundida que es otra unidad defensiva Zerg importante.

La reducción de influencias de las unidades se debe a que al posee valores más altos que los de los edificios (el objetivo final del agente), se daba prioridad a las zonas donde estaban o habían pasado recientemente las unidades que a la base enemiga. La reducción de esta influencia permite, por tanto, una mejor visualización para el agente de las bases del enemigo.

4.3. Experimentación con la IA del juego

Este apartado contiene toda la experimentación realizada para cada versión del agente, analizando los resultados obtenidos.

Las características que se van a evaluar dependiendo de la versión se corresponden con las tablas resumen del final de cada partida:

- **Recursos**: Contempla tres categorías:
  - Mineral: Se corresponde con todo el mineral recolectado por el agente durante la partida.
  - Gas: Se corresponde con todo el gas vespeno recolectado por el agente durante la partida.
  - Gastado: Total de recursos (mineral y gas vespeno) gastados por el agente durante la partida.
- **Unidades**: Contempla tres categorías:
  - Entrenadas: Total de unidades aliadas entrenadas a lo largo de la partida.
  - Perdidas: Total de unidades aliadas eliminadas por el enemigo.
  - Eliminadas: Total de unidades enemigas eliminadas por unidades aliadas a lo largo de la partida.
- **Edificios**: Contempla tres categorías:
  - Construidos: Total de edificios aliados construidos durante la partida.
  - Perdidos: Total de edificios aliados eliminados por el enemigo.
  - Eliminados: Total de edificios enemigos eliminados por unidades aliadas a lo largo de la partida.

Las gráficas mostradas a continuación para cada versión se han generado con los datos obtenidos tras realizar, al menos, dos partidas contra cada raza en cada uno de los mapas indicados en 4.1. Mapas utilizados. Generando la media de todos los valores obtenidos para cada categoría.

Tarea / Versión	Construir edificios	Entrenar unidades básicas	Atacar	Defensa	Gestionar expansiones	Detectar unidades invisibles	Entrenar unidades avanzadas	Investigar mejoras
Versión 0	✔	✔	✔	✔	✘	✘	✘	✔
Versión 1	✔	✔	✔	✔	✘	✘	✘	✔
Versión 2	✔	✔	✔	✔	✘	✘	✘	✔

Versión 3	✔	✔	✔	✔	✔	✔	✔	✔
Versión 4	✔	✔	✔	✔	✔	✔	✔	✔

Tabla 37 Resumen características de cada versión

En la Tabla 37 Resumen características de cada versión se puede observar en:

- Color verde y con un tick: características implementadas y funcionales al 100%.
- Color amarillo y con un tick: características implementadas pero que contienen errores o necesita más trabajo.
- Color rojo y con una equis: características no implementadas.

Todos los datos recopilados se encuentran en 8.3. Resultados de las pruebas.

4.3.1. Versión 0

Esta versión del agente se corresponde con el agente inicial, es decir, el desarrollado al final de Inteligencia Artificial en la Industria del Entretenimiento. Posee un único árbol de comportamiento donde se engloban todas las tareas. Como se toma esta versión como referencia, en las pruebas se evalúan todos los aspectos: Recursos, Unidades y Edificios.

Todas las pruebas realizadas se pueden encontrar en este enlace.

De esta primera versión se van a tomar las puntuaciones obtenidas como referencia para posteriores versiones.

Recursos

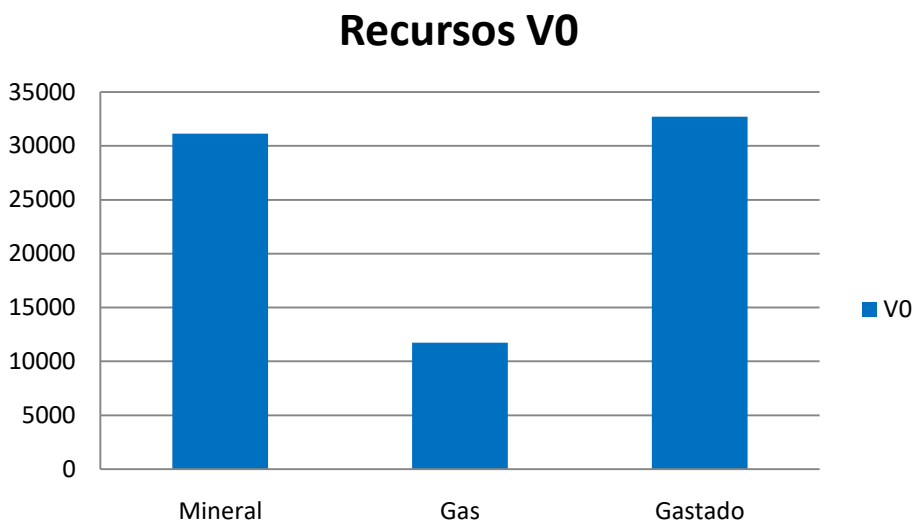


Ilustración 55 Recursos Versión 0

La Ilustración 55 muestra las puntuaciones obtenidas para recursos en la versión 0. Al tratarse de la versión base no se puede realizar una comparación ni análisis en profundidad, ya que los recursos por si mismos no dan mucha información sobre la partida.

Unidades

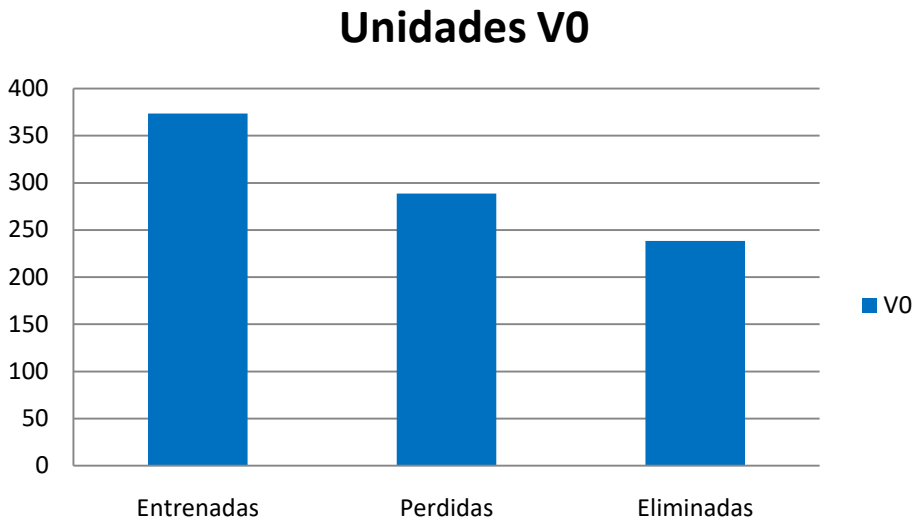


Ilustración 56 Unidades Versión 0

La Ilustración 56 muestra las puntuaciones obtenidas para unidades en la versión 0. A diferencia de los recursos, las unidades si nos permiten ver cómo juega el agente, un punto a destacar es que las unidades perdidas son menos que las unidades entrenadas, por lo que se puede deducir que **el agente gana partidas**, pese a tener versiones rudimentarias de ataque y defensa. Por otro lado, al tener más unidades perdidas que eliminadas podemos deducir también que el agente **pierde más partidas de las que gana**.

Edificios

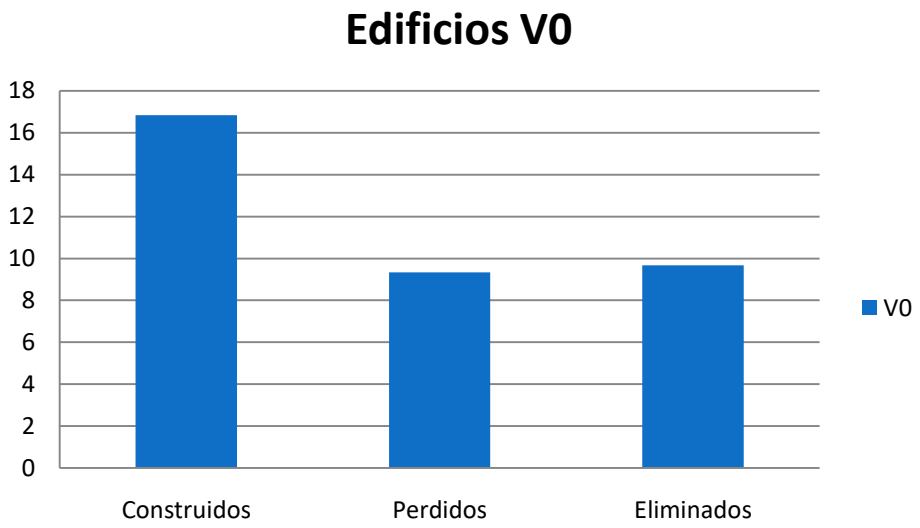


Ilustración 57 Edificios Versión 0

La Ilustración 57 muestra las puntuaciones obtenidas para edificios en la versión 0. Estos valores van acorde a las unidades. Se puede observar cómo el agente pierde de media la mitad de los edificios que construye. Si se observan los datos concretos en 8.3. Resultados de las pruebas sobre cada mapa se aprecia que donde el agente pierde más veces es en los mapas con múltiples entradas a la base, ya que no contempla esa situación en su código.

4.3.2. Versión 1

Esta versión del agente los árboles de comportamiento de la versión 0 se encuentran separado en varios árboles distintos de forma que cada uno se ejecute individualmente. Además, el árbol de construcción se ha mejorado y ajustado.

En este caso las pruebas se han realizado sólo contra la raza *Terran*, ya que aquí se evalúa principalmente la mejora en el orden y construcción de los edificios.

Al modificar solamente el árbol de construcción, carece de sentido evaluar recursos o unidades, ya que sus comportamientos son idénticos. Se compararán los resultados obtenidos en "Edificios" con los de la versión 0 para analizar si los cambios han tenido efectos positivos en el resultado.

Todas las pruebas realizadas se pueden encontrar en este enlace.

Edificios

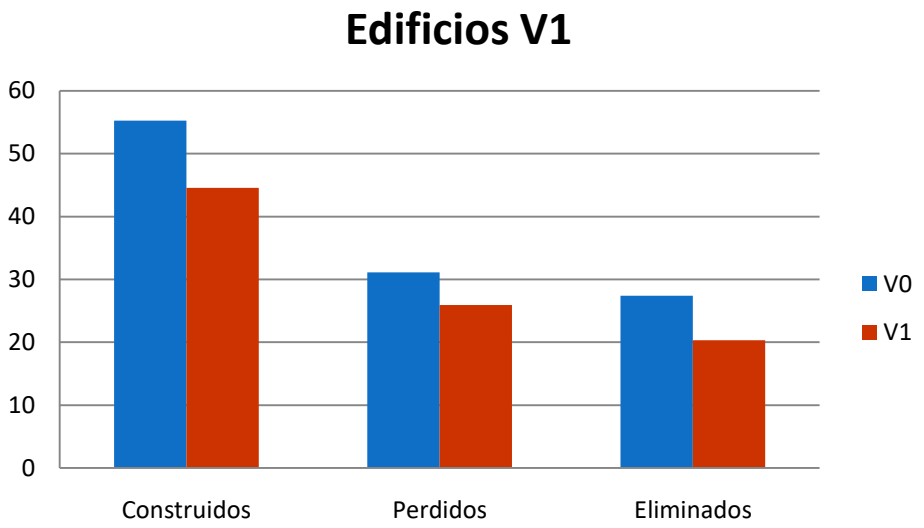


Ilustración 58 Edificios Versión 1

La Ilustración 58 muestra las puntuaciones obtenidas para edificios en la versión 1. Los resultados no acompañan porque en las pruebas realizadas, las partidas han resultado ser peores para el agente: le atacaban antes, las unidades aliadas bloqueaban caminos… Es el motivo por el que los resultados son peores. Sin embargo, esta versión ayudará sobre todo a versiones posteriores, ya que los cambios realizados no se pueden apreciar a simple vista.

Debido a la aleatoriedad existente a la hora de escoger el comportamiento de la IA enemiga, no se puede realizar una experimentación ideal como es en este caso.

4.3.3. Versión 2

Esta versión del agente posee una verdadera gestión de ataque y defensa gracias a la clase "*Troop*". Se prueba solamente contra la raza Terran debido a que es la única que no posee invisibles al comienzo de la partida, lo que permite ver si la gestión realizada para el ataque y defensa es eficiente y útil, ya que está orientado principalmente para partidas largas y no tácticas de tipo *Rush*. Si contemplásemos tropas invisibles no servirían las pruebas ya que siempre se perdería la partida al no tener incluido en esta versión el entrenamiento de unidades de tipo Detector.

Se compararán los resultados obtenidos en "Unidades" y "Edificios" con los de la versión 0 para analizar si los cambios han tenido efectos positivos en el resultado. No se comparan los recursos ya que no se han realizado cambios que afecten directamente a los mismos.

Todas las pruebas realizadas se pueden encontrar en este enlace.

Unidades

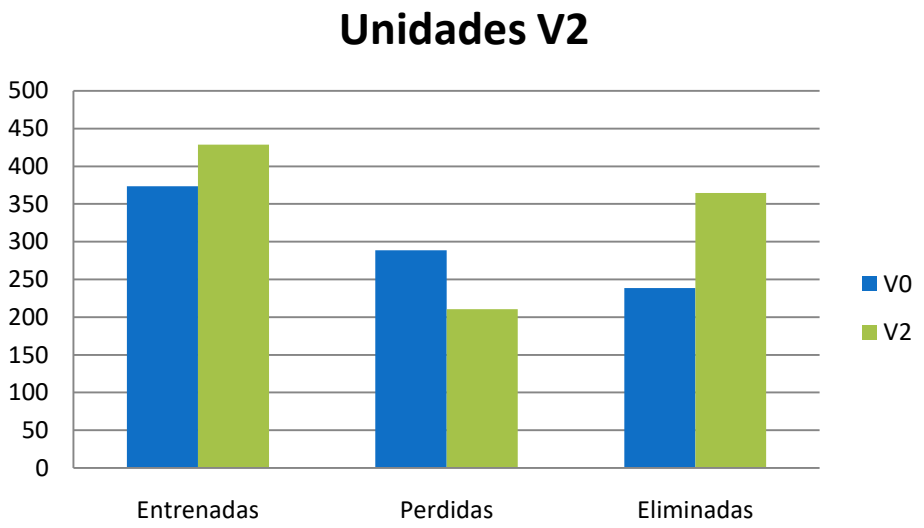


Ilustración 59 Unidades Versión 2

La Ilustración 59 muestra las puntuaciones obtenidas para unidades en la versión 2. En estos resultados se puede apreciar una mejora considerable en todos los aspectos, ya que al tener una gestión optimizada para el ataque y defensa permite que las partidas sean más largas. Como consecuencia, se entrenan más unidades y por tanto, hay más posibilidades de conseguir la victoria.

Se puede ver como el número de unidades perdidas es la mitad de las unidades producidas, mientras que las unidades eliminadas aumentan. Esto se debe a las estrategias de retirada y agrupamiento implementadas, que evitan perder tropas de forma innecesaria y a la larga, ahorrar recursos y disponer de una mayor fuerza de ataque. Como consecuencia directa, significa un número mucho mayor de victorias.

Edificios

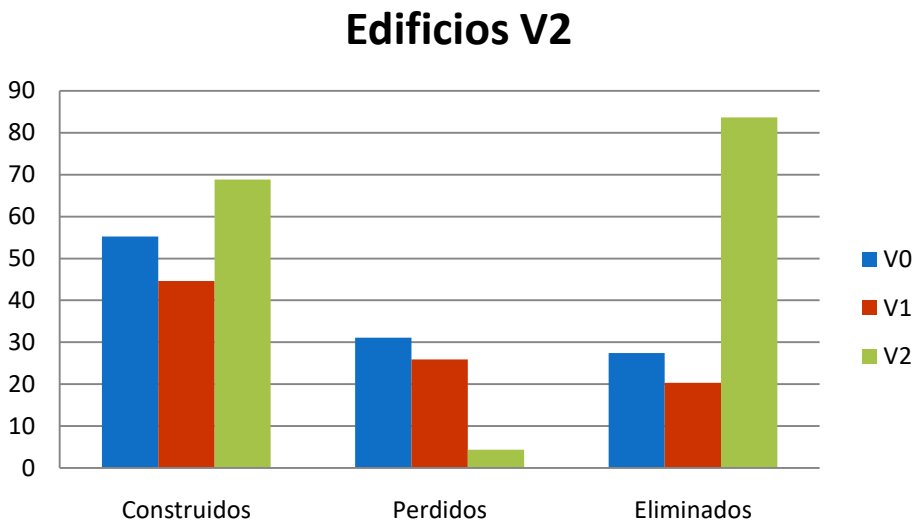


Ilustración 60 Edificios Versión 2

La Ilustración 60 muestra las puntuaciones obtenidas para edificios en la versión 2. Estos valores están directamente relacionados con las unidades y los resultados obtenidos gracias al ataque y defensa. Una mejor defensa implica menos edificios perdidos (prácticamente nulas las pérdidas) y un mejor ataque un mayor número de edificios destruidos (el triple que en la versión 0). Así mismo, el perder pocos edificios permite centrar los recursos en entrenar unidades, construir edificios más avanzados y obtener más suministros, para así entrenar un número mayor de unidades y finalmente: **ganar un mayor número de partidas**

4.3.4. Versión 3

Esta versión del agente sigue mejorando el árbol de construcción, permitiendo construir edificios y unidades hasta ahora no disponibles y en algunos casos, esenciales para poder ganar la partida. Además, contiene una gestión básica de las expansiones, para poder aumentar el número de recursos recolectados.

Se puede considerar una versión "completa" por lo que se comparará con la versión 0 para analizar los resultados.

Todas las pruebas realizadas se pueden encontrar en este enlace.

Recursos

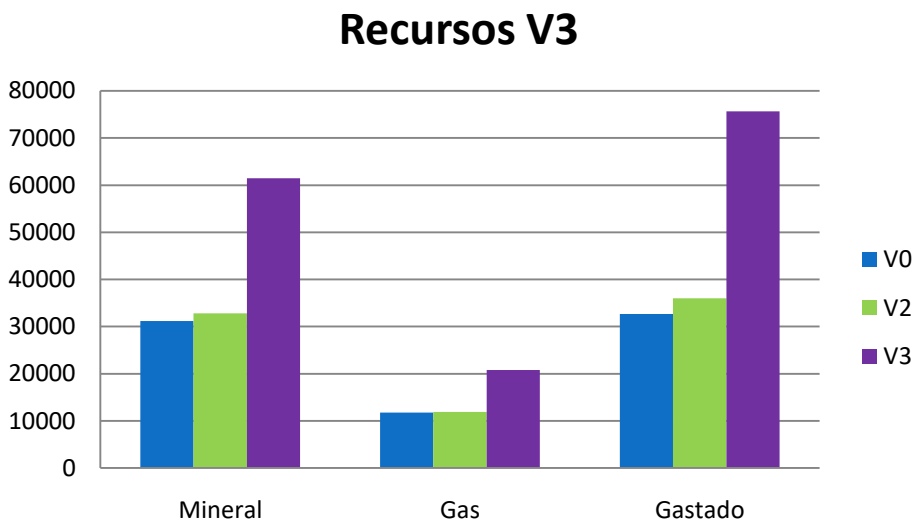


Ilustración 61 Recursos Versión 3

La Ilustración 61 muestra las puntuaciones obtenidas para recursos en la versión 3. La duplicación de los recursos obtenidos tiene dos motivos principales: partidas más largas gracias a las mejoras realizadas en la versión 2 e implementación de la gestión de expansiones para recolectar más recursos.

Como se puede observar, se han duplicado las cantidades recolectadas de cada recurso: más recursos permiten más unidades, más edificios y por tanto, más posibilidades de ganar la partida.

Unidades

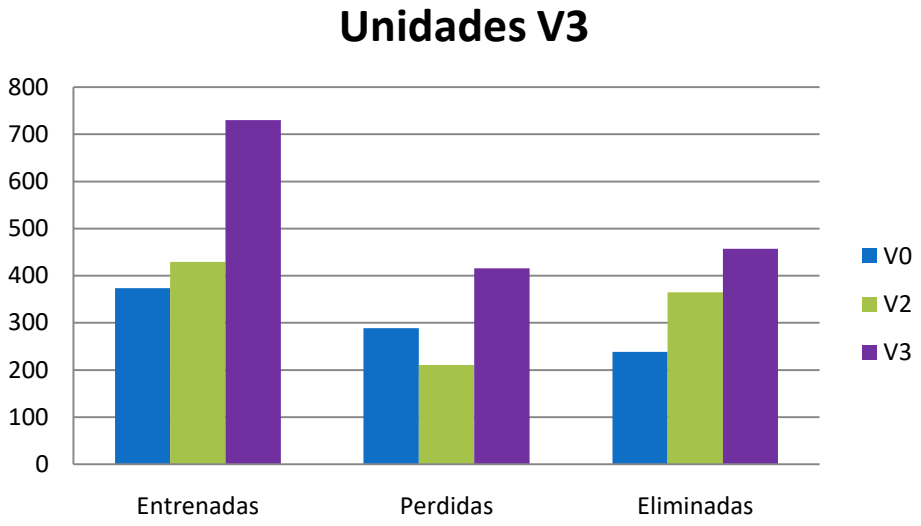


Ilustración 62 Unidades Versión 3

La Ilustración 62 muestra las puntuaciones obtenidas para unidades en la versión 3. El número de unidades entrenadas casi se duplica con respecto a la versión

anterior gracias al mayor número de recursos. Al haber entrenado un mayor número de unidades es lógico que el número de unidades perdidas también se vea afectado. También como consecuencia de partidas más largas y los cambios aplicados en la actualización de la influencia aumenta el número de unidades enemigas eliminadas.

Edificios

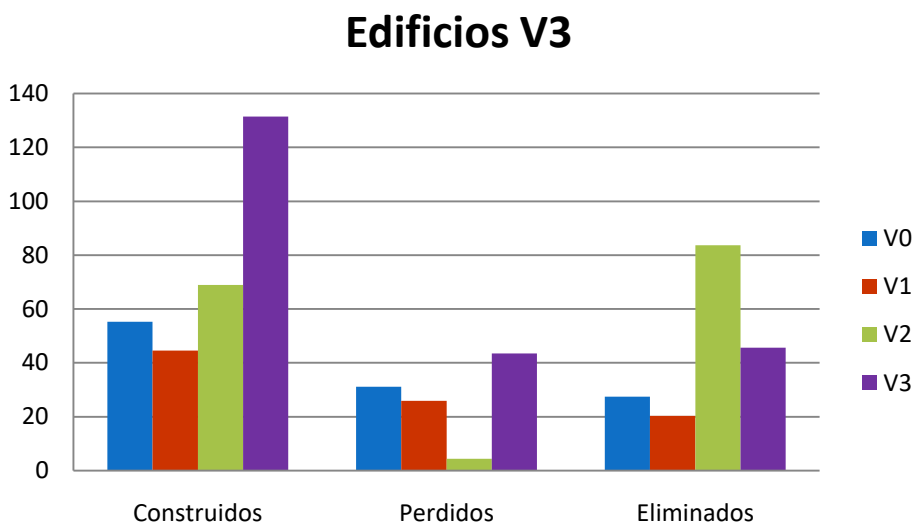


Ilustración 63 Edificios Versión 3

La Ilustración 63 muestra las puntuaciones obtenidas para edificios en la versión 3. De igual forma que con las unidades, un mayor número de recursos y partidas más largas implica poder construir un mayor número de edificios.

El aumento significativo de edificios perdidos es consecuencia de crear expansiones. Al expandir, se construyen edificios en zonas menos protegidas que la base inicial y, cuando el enemigo ataca, suele atacar primero estas zonas, lo que conlleva que pueda destruir más edificios que en la versión 2, donde no había expansiones. Adicionalmente, en mapas como *Astral Balance* o *Breaking Point*, si el enemigo se expandía en zona elevada, el agente no podía alcanzarle, por lo que la partida se alargaba el máximo posible y finalmente perdía la partida (mientras más larga la partida, mayor número de edificios).

La reducción de edificios eliminados se debe a que en las pruebas realizadas, cuando el agente ganaba el enemigo no había construido un gran número de edificios.

4.3.5. Versión 4

Esta versión del agente contempla contra el tipo de raza que se juega. Dependiendo del enemigo, se le da prioridad a ciertos elementos o a otros.

Al igual que la versión 0 y 3, se puede considerar un agente "completo" y se compararán sus resultados con los obtenidos en las versiones 0 y 3.

Todas las pruebas realizadas se pueden encontrar en este enlace.

Recursos

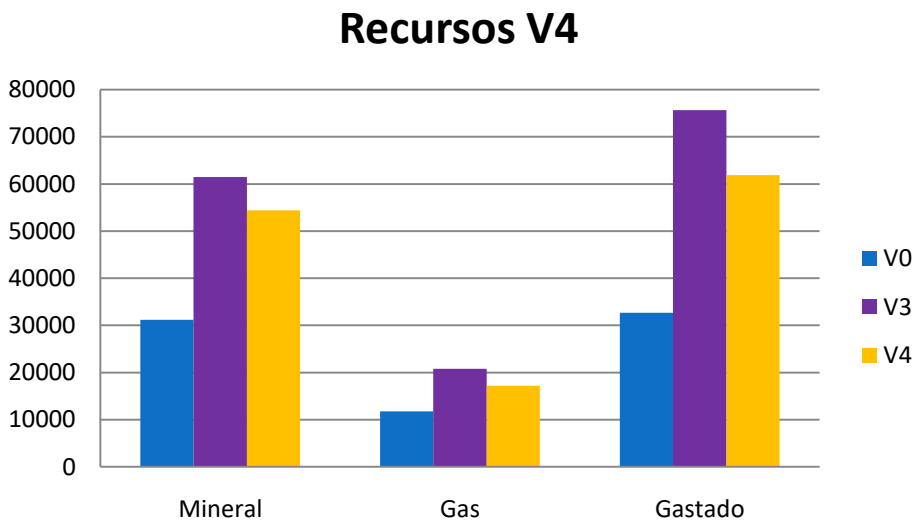


Ilustración 64 Recursos Versión 4

La Ilustración 64 muestra las puntuaciones obtenidas para recursos en la versión 4. Los recursos en esta última versión siguen el camino marcado por la versión 3, ya que no se ha realizado ningún cambio significativo. La reducción de los valores se debe principalmente a partidas más cortas.

Unidades

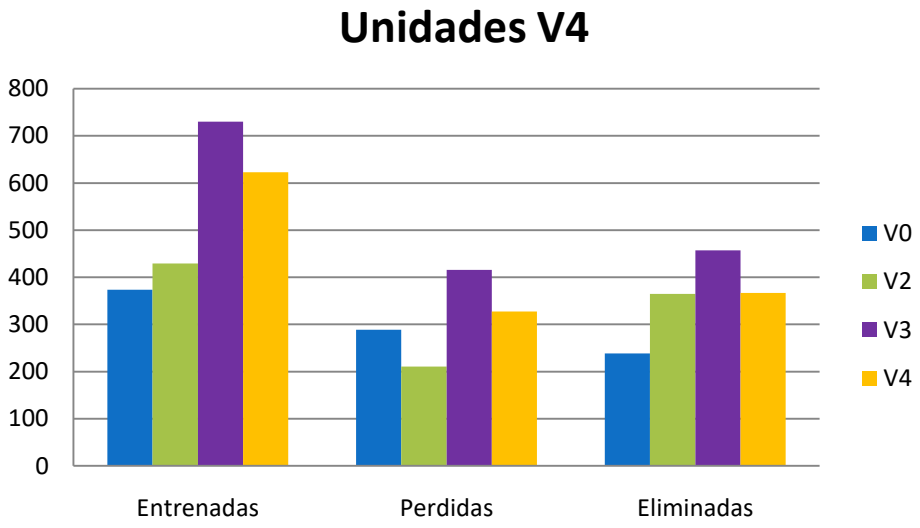


Ilustración 65 Unidades Versión 4

La Ilustración 65 Ilustración 66muestra las puntuaciones obtenidas para unidades en la versión 4. De igual forma que los recursos, las partidas realizadas fueron algo más cortas que las anteriores, por lo que las unidades se resienten de forma similar.

Edificios

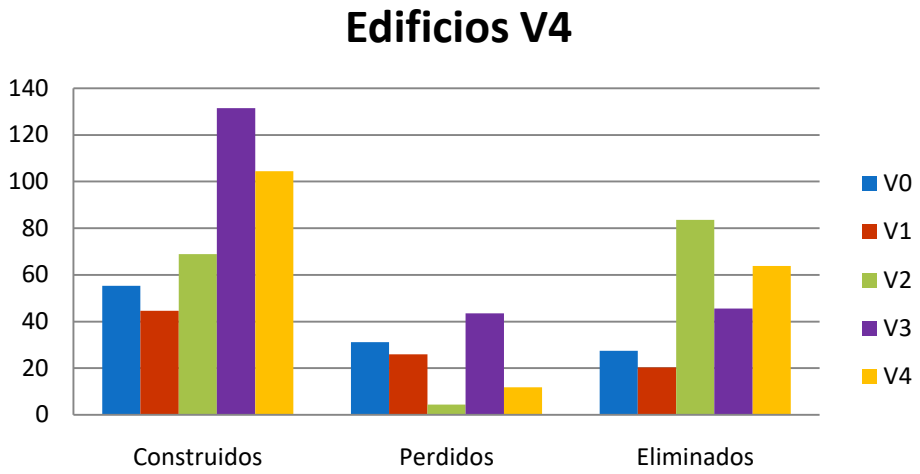


Ilustración 66 Edificios Versión 4

La Ilustración 66 muestra las puntuaciones obtenidas para edificios en la versión 4. En los edificios se puede apreciar un cambio significativo en el número de edificios perdidos y en un aumento de los edificios destruidos. Esto tiene como consecuencia principal la nueva influencia aplicada, que permite un reconocimiento mucho mejor de las posiciones del enemigo y permite realizar ataques más efectivos y al mismo tiempo, evitar que lleguen a la base. De igual forma, las partidas más cortas se deben también a este cambio, ya que al atacar mejor y eliminar casi por completo los bloqueos de movimiento, las victorias ocurren antes.

Capítulo 5: Gestión del proyecto

En este apartado se explica el proceso seguido para el desarrollo del agente. Describiendo la metodología usada, la planificación resultante y análisis de la misma.

5.1. Descripción de las fases del proyecto

Al tratarse de un desarrollo *software* se ha seguido una metodología de desarrollo ágil durante el proyecto. Éstas se caracterizan por seguir un desarrollo incremental e iterativo, lo cual implica que en cada iteración se ha de realizar una planificación, análisis de requisitos, codificación y pruebas. Existen multitudes de metodologías ágiles, siendo la elegida para el proyecto *Scrum* [Schwaber, 2004].

La metodología *Scrum* se caracteriza por tener los siguientes roles:

- *Product Owner*: Representa al cliente y es el encargado de formalizar las tareas (Historias de usuario) a realizar y darles prioridad. Sólo existe un único *Product Owner* por equipo.
    - Una historia de usuario consiste en una representación breve (dos o tres frases) sobre el requisito que se desea implementar. Cada historia de usuario debería poder escribirse en un *post-it*.
- Equipo de desarrollo (o *Development Team*): Son los encargados de realizar las tareas e ir entregando al final de cada iteración versiones funcionales e incrementales del producto.
- *Scrum Master*: Es el encargado de supervisar el desarrollo del producto y asegurar de que se alcanzan las metas acordadas. Para ello realiza tareas como: facilitar las reuniones diarias o con el *product owner*, motivar la auto-organización del equipo de desarrollo y ayudarles para evitar bloqueos en el desarrollo y así no detener el progreso.

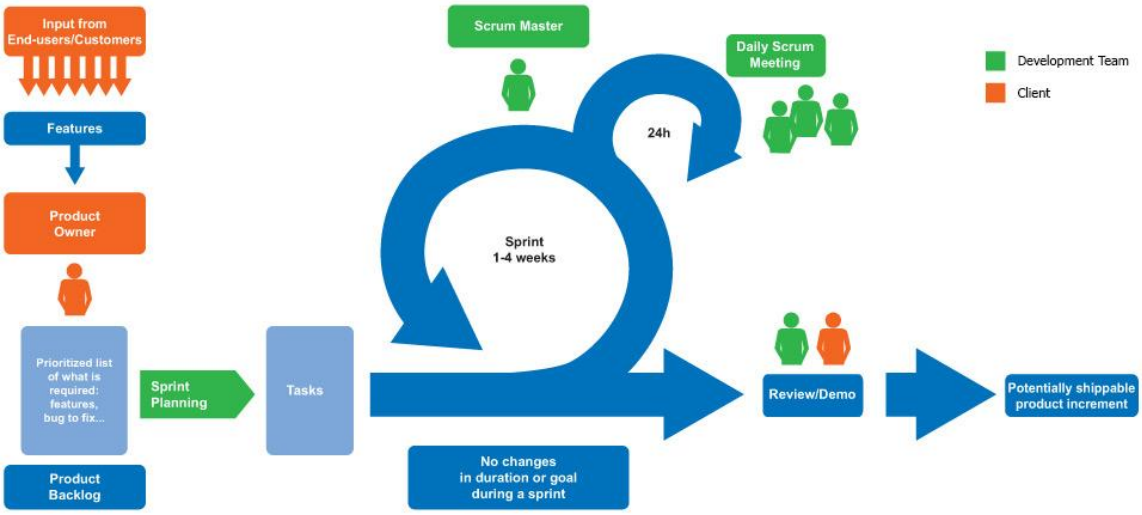


Ilustración 67 Ciclo de vida de Scrum (Fuente

Cada iteración se llama *Sprint*, que suele durar entre 2 y 3 semanas. Dentro del mismo existen las siguientes fases (Ilustración 67):

1. Planificación: Se realiza al comienzo del *sprint.* En esta reunión se decide cual es el objetivo u objetivos a alcanzar y se dividen en tareas más pequeñas. Una vez divididas cada miembro del equipo de desarrollo escoge las tareas que va a realizar en ese *sprint*.
2. Reunión diaria (o *Daily Scrum*): Se trata de una reunión corta (a lo mucho 15 minutos) que se realiza todos los días, a la misma hora, en el mismo lugar y estando de pie. En esta reunión cada miembro del grupo explica qué hizo el día anterior y qué piensa hacer hoy. Si se encuentra con algún problema que le impida avanzar lo debe comunicar al resto del grupo.
3. Demo: Se realiza al finalizar el *sprint.* En esta fase se muestra al *product owner* el trabajo realizado. De esta forma se pueden detectar problemas por falta de entendimiento entre el *product owner* y el equipo de desarrollo o el cliente.
4. Retrospectiva: Se realiza al finalizar el *sprint*. En esta reunión se busca sobre todo encontrar problemas que hayan surgido durante el *sprint* con el objetivo de solucionarlos y mejorar de cara a próximos.

Dado que en este proyecto no era posible seguir la metodología *Scrum* completamente ya que todos los roles recaen en una única persona, lo que se ha seguido ha sido su planteamiento de flujo de trabajo: realizar *sprints* de un mes de duración, dividir el trabajo en subtareas e ir realizando versiones incrementales del agente. Las fases resultantes de esta planificación han sido:

- Versión 1 (Enero): En este *sprint* se mejora el agente separando los árboles de ataque, construcción y entrenamiento, de forma que sean

independientes unos de otros. Además se mejora la forma en la que se construyen los edificios y encuentran zonas libres para construir.

- Versión 2 (Febrero): En este *sprint* se creará un árbol para gestionar ataque de las unidades y defensa de la base.
- Versión 3 (Marzo): En este *sprint* se seguirá mejorando el árbol de construcción, contemplando nuevas situaciones y añadiendo una gestión de las expansiones, de forma que se obtengan más recursos.
- Versión 4 (Abril): En este *sprint* se contemplarán distintas estrategias dependiendo de la raza que sea el enemigo.

5.2. Planificación

En este apartado se detalla la realización de cada *sprint*, con las tareas desarrolladas y duración de las mismas. Como se ha comentado en el apartado anterior, cada *sprint* tiene una duración aproximada de 1 mes.

Al ser un proceso iterativo, la finalización de un *sprint* no prohíbe modificar y ajustar valores o arreglar errores de las tareas realizadas en el mismo. Estos ajustes se pueden apreciar principalmente en la influencia y el ataque y defensa mediante tropas, que a partir de la versión 2 han seguido sufriendo ajustes y mejoras.

Para su correcta visualización se muestra a continuación cada diagrama Gantt, correspondiente a la planificación de cada versión:

5.2.1. Planificación de la Versión 1

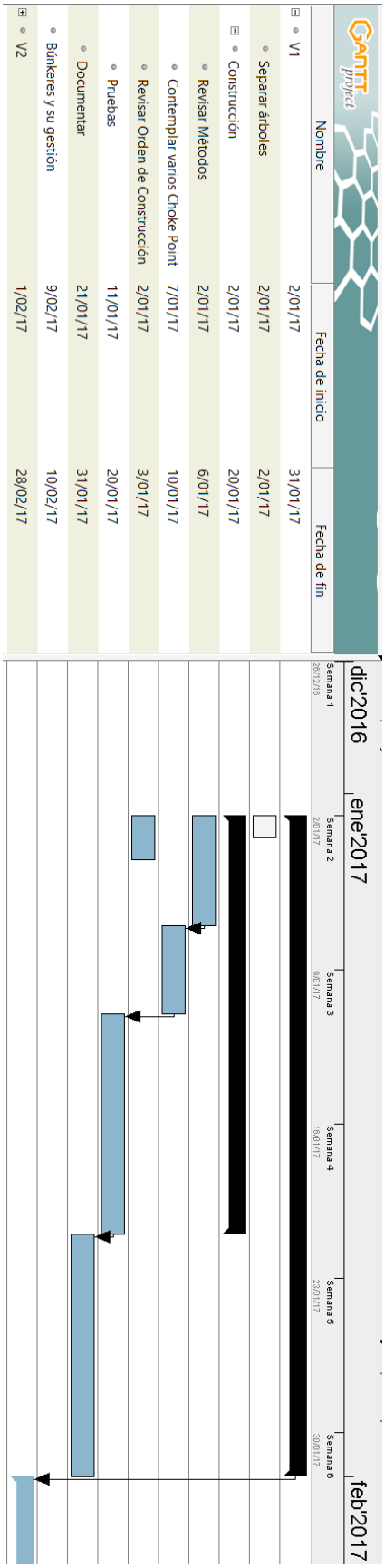


Ilustración 68 Gantt V1

5.2.2. Planificación de la Versión 2

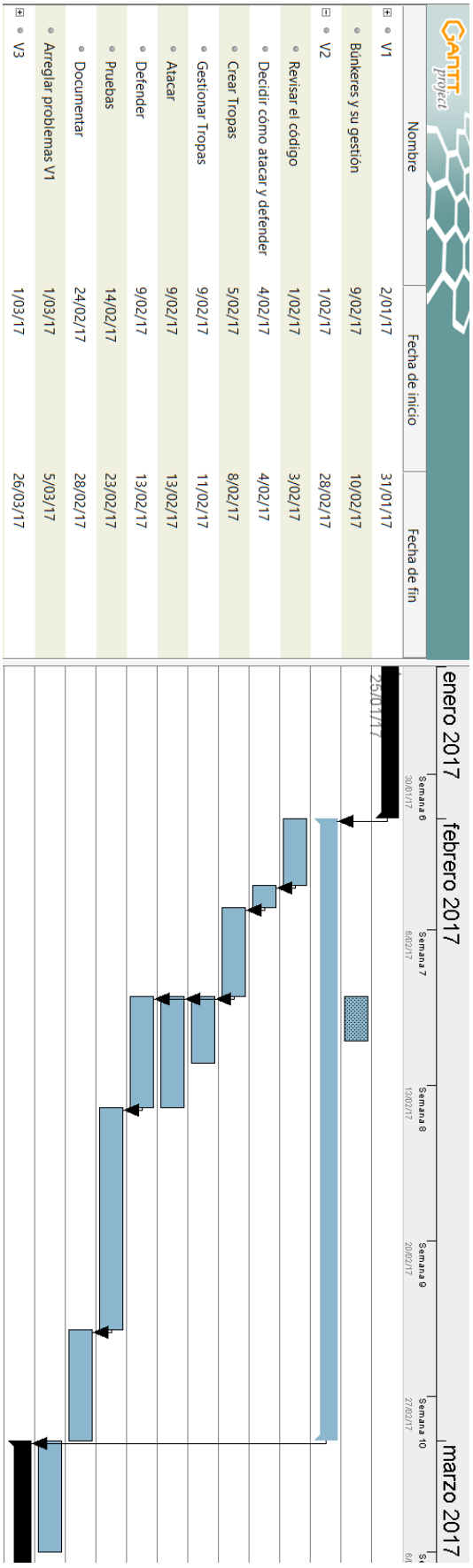


Ilustración 69 Gantt V2

5.2.3. Planificación de la Versión 3

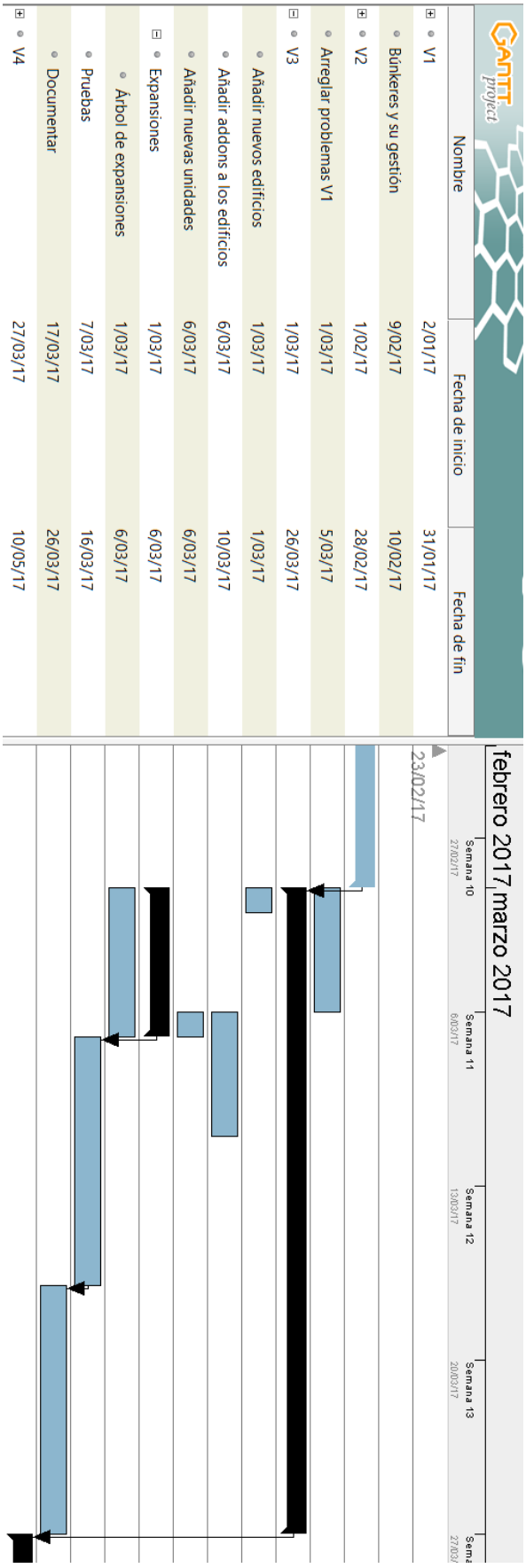


Ilustración 70 Gantt V3

5.2.4. Planificación de la Versión 4

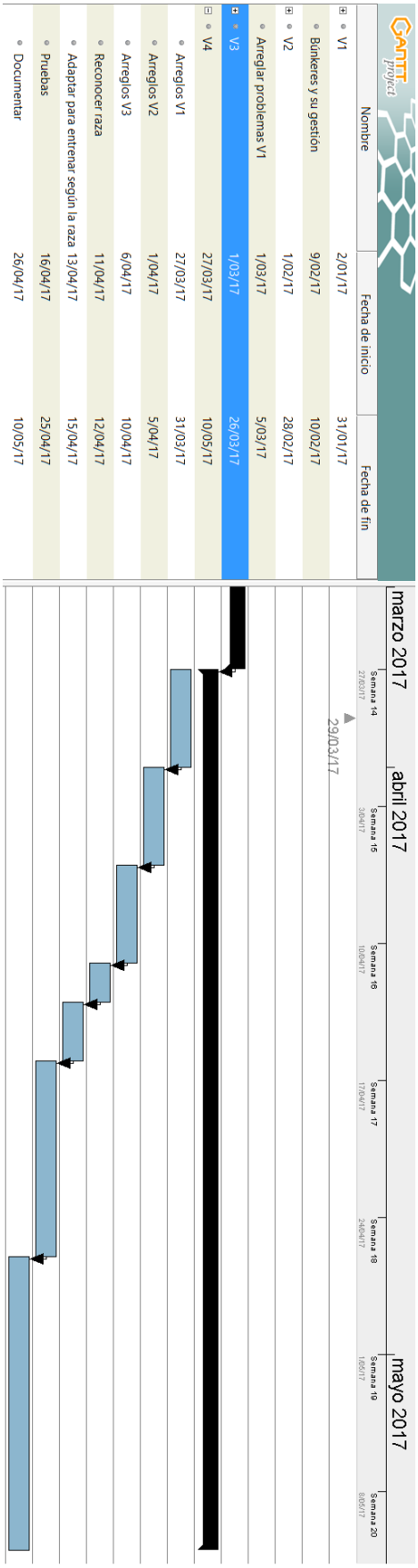


Ilustración 71 Gantt V4

5.3. Presupuesto

Para el desarrollo del proyecto hay que contar tanto con material fungible, como software y personas.

El tiempo de desarrollo han sido 7 meses. Como costes indirectos se ha considerado un 15% del coste total. Estos costes son referidos a aquellos no específicos del proyecto, como puede ser agua, luz, internet o transporte.

5.3.1. Material fungible

La fórmula seguida para la amortización ha sido:

$$\left(\frac{M}{V}\right) * C$$

- **M**: Número de meses de uso del material.
- **V**: Número de meses de vida estimados.
- **C**: Coste del material en cuestión.

El material fungible necesario para el desarrollo del proyecto consta de un ordenador de sobremesa con las siguientes especificaciones:

- **Procesador**: Intel Core i5-4460 3.2Ghz.
- **Tarjeta Gráfica**: MSI GeForce GTX 750 Ti LP 2GB GDDR5.
- **Placa base**: Gigabyte GA-H81M-S2H.
- **Memoria RAM**: Kingston HyperX Fury Blue DDR3 1600 PC3-12800 8GB CL10.
- **Disco duro**: WD Black 500GB.
- **Teclado y ratón**: Logitechnology Wireless Combo MK270.
- **Caja**: Nox Hummer MC USB 3.0 Zero Edition Blanca.
- **Fuente de alimentación**: Tacens Mars Gaming 800W 80 Plus.

Nombre	Coste (€)	Amortizado	Total
Intel Core i5-4460 3.2Ghz	179,00€	144,19€	34,81€
MSI GeForce GTX 750 Ti LP 2GB GDDR5	119,00€	95,86€	23,14€
Gigabyte GA-H81M-S2H	52,00€	41,89€	10,11€
Kingston HyperX Fury Blue DDR3 1600 PC3-12800 8GB CL10	61,75€	49,74€	12,01€

Nombre	Coste (€)	Amortizado	Total
WD Black 500GB	59,00€	47,53€	11,47€
Logitechnology Wireless Combo MK270	24,75€	19,94€	4,81€
Nox Hummer MC USB 3.0 Zero Edition Blanca	49,00€	39,47€	9,53€
Tacens Mars Gaming 800W 80 Plus	53,50€	43,10€	10,40€
Total	598,00€	481,72€	116,28€

Tabla 38 Costes de material fungible

5.3.2. Software

Los programas necesarios para poder desarrollar todo el proyecto son:

- Java: Lenguaje de programación diseñado por Oracle. Se trata de un lenguaje de propósito general y orientado a objetos. Es debido a que la API utilizada está escrita en este lenguaje.
- Microsoft Word y Microsoft PowerPoint: Programas pertenecientes a la *suite* de Microsoft Office. Necesarios para el desarrollo de la documentación y presentación del trabajo.
- Eclipse: Entorno de desarrollo integrado (IDE) para el lenguaje Java principalmente. En este IDE se desarrolla todo el agente.
- JNI-BWAPI: *Wrapper* en Java de la API BWAPI escrita en C++. Es la API que se va a utilizar para el desarrollo del agente.
- *ChaosLauncher*: Aplicación que sirve para conectar el agente desarrollado con el videojuego *StarCraft*.
- *StarCraft* y *StarCraft: Brood War*: RTS sobre el cual se realiza el trabajo.

Nombre	Coste (€)	Amortizado	Total
Java	0,00€	0,00€	0,00€
Microsoft Office	149,00€	120,03€	28,97€
Eclipse	0,00€	0,00€	0,00€

Nombre	Coste (€)	Amortizado	Total
JNI-BWAPI	0,00€	0,00€	0,00€
ChaosLauncher	0,00€	0,00€	0,00€
Starcraft, Starcraft:Brood War	0,00€	0,00€	0,00€
Total	149,00€	120,03€	28,97€

Tabla 39 Coste de software

5.3.3. Personal

El personal necesario para el correcto desarrollo del trabajo serán: Jefe de proyecto, Programador y Tester.

Categoría	Coste (€) / Hora	Horas dedicadas	Total
Jefe de proyecto	35€	40	1.400€
Desarrollador	25€	200	5.000€
Tester	20€	70	1.400€
	Total	310	7.800€

Tabla 40 Coste de personal

5.3.4. Total

El coste total del proyecto es mil novecientos dieciocho con setenta y siete EUROS:

Concepto	Coste
Personal	7.800€
Material fungible	116,28€
Software	28,97€
Costes Indirectos (15%)	1191,78€
IVA (21%)	1918,77€
Total	**11.055,80€**

Tabla 41 Coste total del proyecto

5.4. Marco regulador

Según se recoge en el Real Decreto Legislativo 1/1996 (Referencia: BOE-A-1996-8930) [21] se entiende como programa de ordenador "toda secuencia de instrucciones o indicaciones destinadas a ser utilizadas, directa o indirectamente, en un sistema informático para realizar una función o una tarea o para obtener un resultado determinado, cualquiera que fuere su forma de expresión y fijación".

Al formar España parte de la Comunidad Europea también se aplican otras leyes a parte de la legislación española, sin embargo, al ser muy similar no se ha considerado relevante para esta definición.

Por lo tanto, el agente desarrollado se debe legislar por todas las leyes relacionadas, siendo las más destacables las de la legislación española, para nuestro caso:

- **Artículos 17. Derecho exclusivo de explotación y sus modalidades:** Este artículo informa de que sólo el autor puede reproducir, distribuir y transformar el programa, a excepción de casos autorizados o protegidos por la Ley.
- **Artículo 21. Transformación: "**La transformación de una obra comprende su traducción, adaptación y cualquier otra modificación en su forma de la que se derive una obra diferente".
- **Artículo 97. Titularidad de los derechos:** El autor del programa será el creador en caso de ser una única persona o empresa, y creadores en caso de ser un grupo o colectivo.
- **Artículo 102. Infracción de los derechos:** Los derechos de autor se infringen principalmente si:
    - Se pone en circulación una o más copias de forma ilegítima.
    - Buscar fines comerciales con una o más copias obtenidas de forma ilegítima.
    - "Quienes pongan en circulación o tengan con fines comerciales cualquier instrumento cuyo único uso sea facilitar la supresión o neutralización no autorizadas de cualquier dispositivo técnico utilizado para proteger un programa de ordenador."

En este caso, de los Artículos 17, 21 y 97 sólo se aplicarían la autoría del programa, ya que el código en sí mismo es *open source* y se encuentra publicado bajo licencia GLPv2 [22], de forma que cualquier código derivado siempre se mantenga *open source*. Por lo que cualquiera puede modificarlo a su placer, siempre que se mantenga la licencia.

Respecto al artículo 102, tiene especial importancia porque al utilizar el inyector de código se están violando los términos de uso (*End User License Agreement*, EULA) de *StarCraft*, pero desde la propia Blizzard han dado el visto bueno a todo el

proyecto relacionado y ha apoyado el proyecto dando premios para el AIIDE [SCAI, 2016].

5.5. Impacto socio-económico

La realización exitosa de este trabajo tendría repercusión principalmente en el campo de la IA y de los videojuegos de estrategia. Al tratarse de una IA para un videojuego carece de impactos sociales, medioambientales o éticos.

Hoy en día la gran mayoría de IAs para juegos de estrategia se realizan mediante Arquitecturas de Pizarra, Comportamientos Orientados a Metas y Redes Bayesianas entre otros. Su uso se debe principalmente a que permiten modelar la incertidumbre presente por la niebla de guerra (Redes Bayesianas) y dirigir el comportamiento hacia una tarea u otra (Pizarra y Orientado a Metas). Sin embargo, la gran mayoría de veces esa incertidumbre puede eliminarse o ignorarse a lo largo de la partida, ya que se toma desde el inicio una estrategia a seguir y el jugador no se arriesga tanto, pues puede cambiar más adelante de estrategia si se obtiene nueva información. Es decir, la niebla de guerra aunque existe y se puede tener en cuenta, al final se puede ignorar.

Durante la realización de este proyecto no se ha encontrado ningún trabajo de IA en RTS creada con Árboles de Comportamiento, por lo que puede servir como primer paso para que más gente piense en utilizar Árboles de Comportamiento como modelos de toma de decisiones en RTS y no sólo en juegos de acción en primera persona dado que los resultados obtenidos son bastante prometedores.

Capítulo 6: Conclusiones y trabajos futuros

En este capítulo se trata en primer lugar las conclusiones generales extraídas de la realización de este trabajo, problemas encontrados y lo que me ha aportado a nivel personal. A continuación se desarrollan conclusiones concretas respecto a cada objetivo definido inicialmente en 1.3 Objetivos del trabajo y finalmente se proponen mejoras a realizar sobre el agente desarrollado.

6.1. Conclusiones generales

El desarrollo de una IA para videojuegos es bastante sencillo, siempre y cuando se tengan que modelar comportamientos simples. Sin embargo cuando se quiere comenzar con comportamientos mucho más complejos, aunque seleccionar la técnica correcta ayude a simplificar todo el proceso, es una tarea compleja, donde la gran parte del tiempo se pasa comprobando los nuevos cambios realizados y procurando que no entren en conflicto con comportamientos previos. Además obliga a codificar de forma eficiente desde el punto de vista de la memoria, pues las respuestas normalmente deben ser en tiempo real o en un espacio muy corto de tiempo, lo que hace que el balance calidad-coste sea bastante importante. Para poder tener accesos rápido, es necesario guardar en memoria un gran número de variables, por lo que también es importante decidir cuál es la información a almacenar durante la partida.

La realización de este trabajo ha permitido continuar el desarrollo del agente codificado en la asignatura "Inteligencia Artificial en la Industria del Entretenimiento". Para así profundizar en más aspectos del mismo y conocer en mayor profundidad el funcionamiento de BWAPI. A raíz de todo este trabajo ha crecido el interés por el SSCAIT [SSCAIT, 2017]. Además sirve como primera experiencia en el desarrollo de IA para videojuegos, un campo bastante demandado hoy en día.

En cuanto a los problemas encontrados, los más importantes durante el desarrollo han sido:

- Detectar ataques: si bien se puede detectar cuándo un enemigo sale de la niebla de guerra, no es posible saber cuándo una unidad está siendo atacada o está comenzando a atacar, por lo que si se quiere conocer esta información es necesario comprobarla en cada iteración del juego. Esto se resolvió para las unidades contando el número de unidades vivas en las tropas en cada iteración.
- Construcción: la construcción sigue necesitando retoques para funcionar perfectamente, ya que deja mucho espacio vacío sin motivo aparente y gran parte del tiempo dedicado al agente en las primeras versiones consistió en arreglar y probar en profundidad los cambios realizados.

- Movimiento: el uso del mapa de influencia es bastante útil para definir qué posiciones atacar. Sin embargo si se quieren mover las unidades a posiciones no identificadas por el propio juego (expansiones, *choke points*, enemigos…) es bastante complicado. Este problema se producía al agrupar las unidades cuando se separaban mucho. Para solucionarlo, en vez de agruparlas en una posición concreta del mapa, se agrupaban alrededor de una unidad de las que formaba la tropa.

6.2. Conclusiones referentes a los objetivos

A continuación se realizan tres conclusiones relacionadas con cada uno de los objetivos del Capítulo 1: Introducción.

- Análisis del entorno: este objetivo se ha cumplido de forma bastante satisfactoria, realizando estrategias distintas según la zona inicial, construyendo edificios en lugares específicos, haciendo uso del mapa de influencias y construcción y construyendo expansiones en sus lugares correspondientes.
- Toma de decisiones: este objetivo está parcialmente cumplido, en la última versión se han establecido unas listas concretas de unidades a entrenar según el enemigo al que nos enfrentemos y si el enemigo genera unidades invisibles se vuelve una prioridad crear una nave científica para detectar a los invisibles. Sin embargo, todavía queda pendiente definir más estrategias posibles a realizar.
- Gestión de ataque y defensa: este objetivo se ha cumplido satisfactoriamente, gracias a la división de las unidades e implementación de las tropas con estados, se puede tener una gestión del total de unidades militares, pudiendo ordenarles donde situarse y qué acciones realizar. Aún así, se puede seguir mejorando para tener un comportamiento más complejo.

6.3. Trabajos futuros

A la última versión del agente desarrollado todavía le quedan una serie de mejoras a realizar para poder dar por "concluido" el agente. Gran parte de ese trabajo pendiente consiste en añadir técnicas y comportamientos que pueden ser necesarios para garantizar la victoria en muchas partidas.

En cuanto a las técnicas, las más importantes y, en cierto modo, urgentes son:

- Establecer medidas para evitar el bloqueo del movimiento de las unidades por los cuellos de botella. Esto podría realizarse teniendo en cuenta el tiempo que lleva la unidad sin moverse, o si el estado de reagrupar no varía en un periodo concreto.
- Poder cargar y descargar unidades en zonas elevadas para así poder alcanzar zonas inaccesibles. Una buena forma de comenzar a definir estos

comportamientos sería dedicar un árbol específico para comportamientos "extras" tales como utilizar habilidades o mover unidades concretas como las naves de evacuación, cargar y descargar sus unidades.

- Realizar una gestión de microacciones a la hora de atacar con ciertas unidades, ya que por ejemplo, los buitres son especialmente efectivos y devastadores cuando se hace una buena micro con ellos. La forma más común de realizar esta acción es fijar un enemigo y sólo atacarle cuando se encuentre en el límite del rango de ataque y si el enemigo se acerca, alejarse de él hasta que vuelve a situarse en el rango.

Respecto al comportamiento de ataque y defensa, un aspecto importante que se podría implementar es un análisis de los enemigos cercanos, para así ser capaces de predecir si se ganaría un combate o centrar el ataque a unidades específicas, como los médicos.

Finalmente otro aspecto importante es pasar de JNI-BWAPI a BWMirror, si se quiere continuar con Java como lenguaje de programación o directamente codificar todo el agente en C++ usando directamente la API. Esto permitirá utilizar la versión más actualizada y aprovecharse así del arreglo de *bugs* y mejora de eficiencia de las últimas versiones.

Chapter 7: Design and development of behaviors in StarCraft: Brood War

7.1. Introduction

Videogames have become one of the pillars in the entertainment industry, becoming its first source of profit. Within all the existents genres, it can be found Real Time Strategy (RTS). RTS games are distinguished for making players to play against each other in a closed map, where they perform their actions simultaneously, without turns and competing to achieve certain goal, e.g.: destroy the enemy base, watch and control certain zone during a specific time or gather certain amount of resources. Other aspect to be highlighted in this genre is its complexity, forcing players to do and manage multiple actions and different situations during the game. The players must adapt their strategies in every match. In addition, when considering all the game machinery that has been built in this kind of games, they become a perfect environment for development and testing of AI techniques such as path finding, optimization or militar behaviors.

Inside existent RTS, it can be found *StarCraft*, launched in 1998 and still played nowadays. It is one of the most popular and with large number of players RTS game in history. *StarCraft* has become very famous in entertainment, competition and research due to three main features: big maps with a lot of elements, management of large amount of units and three completely different races. The most important feature is the last one. Every race has unique traits, making them play and strategies in different ways. This forces to adopt different approaches and strategies, according to the map and the selected and enemy race.

In order to offer a true challenge to players, automatic players or *bots* should be able to adapt to the current situation in the games, which involves considerering a large number of parameters, which may be their own, enemy's or enviroment's. However, reality is very different, where instead of making *bots* which can adapt, they have hardcoded a fixed list of behaviors and one of them is selected at the beginning of the game, no matter how the other player plays.

A possible solution to this problem can be found by making an approach with Machine Learning techniques such as Artificial Neural Networks, Clustering or Genetic algorithm or with more handmade techniques for instance Behavior Trees, Decision Trees or Finite State Machines.

7.2. Main Objectives

The aim of this work is the development of an intelligent agent using AI techniques. The agent must be able to defeat game's AI and adapt to both, the map and enemy. This means:

- Environmental analysis: in order to be able to take decisions such as: where to build, send troops, gather resources or expand.
- Choose strategies: as explained in the Introduction, each race works totally different from each other.
- Management attack and defense: most important feature, without a good attack and defense it will not be able to win any match.

7.3. Design

In this section the architecture of the bot is described. Explaining its components and functionality.

7.3.1. Global vision

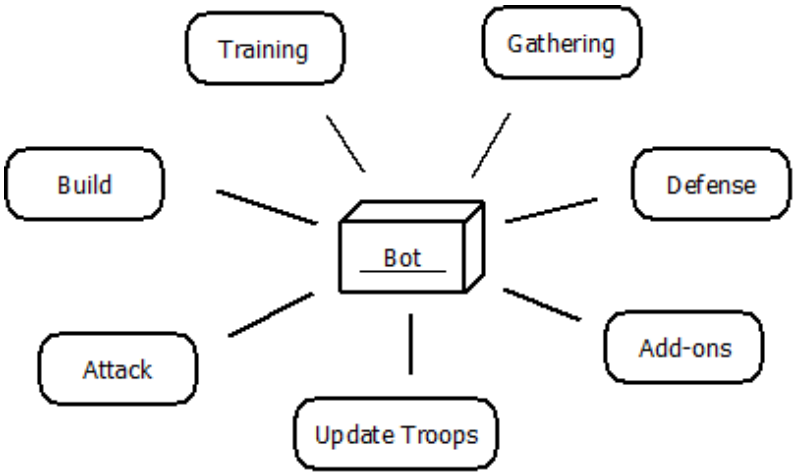


Illustration 72 Global vision of the bot

Every component will be described in detail in the following sections, next a brief description about each component:

- Agent: it is the kernel of the system and is responsible for managing and selecting which action to perform on each frame. JNI-BWAPI's event listeners are implemented in it. It works as a root of a behavior tree where the rest of behaviors trees for managing a specific task are attached as its children.
- Gathering: this behavior tree is responsible for those actions related to workers (SCVs). It is composed by three sequences: *Repair*, *Select CC* and *Gather resources*.
- Training: this behavior tree is responsible for training new units.
- Build: this behavior tree is responsible for constructing new buildings and upgrade units. It is composed by two sequences: *Build Buildings* and *Research*.
- Add-ons: this behavior tree is responsible for building add-ons for the buildings.
- Attack: this behavior tree is responsible of managing the strategy.

- Update Troops: this behavior tree is responsible for creating, updating and maintaining the troops.
- Defense: this behavior tree is responsible for managing the base defenses. It is composed by two sequences: *Bunker* and *Defense*.

On each execution of JNI-BWAPI's *matchFrame* function (equals to BWAPI's *onFrame*) the agent executes each tree, creating the ingame actions.

7.3.2. Agent

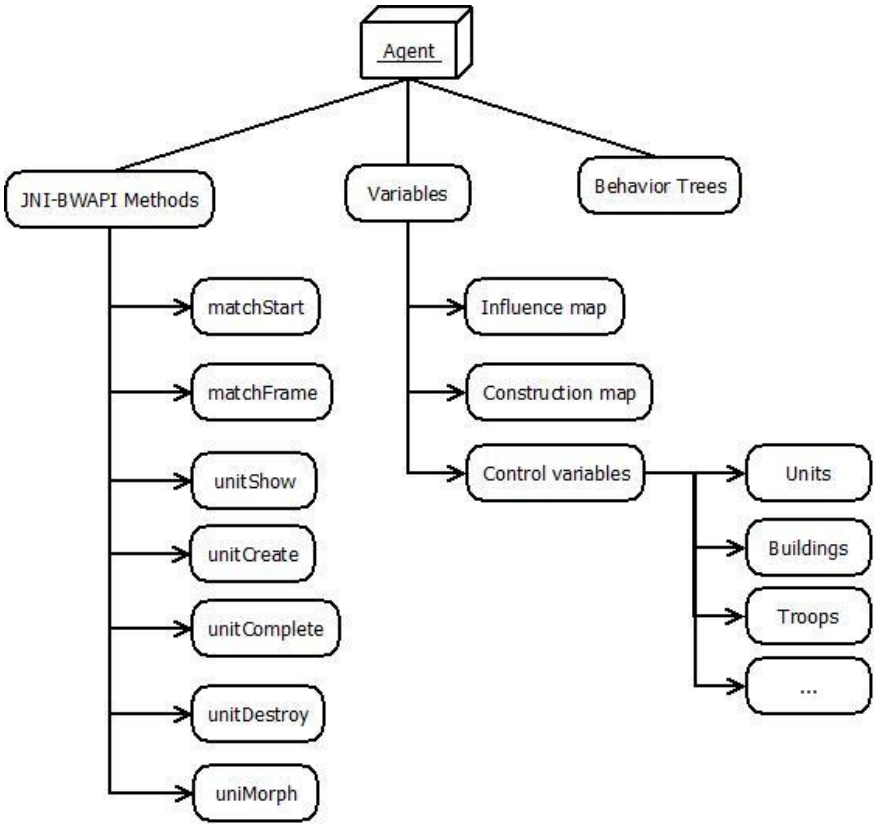


Illustration 73 Agent components

Illustration 73 shows graphically the agent structure. It is composed mostly of Behavior Trees and Variables. Through these elements and with JNI-BWAPI's listeners is made the agent functionality.

Variables are the nexus between the listeners and behavior trees; through them the agent can itself updated information about the match.

- **JNI-BWAPI listeners:**
  - matchStart (equals to BWAPI's onStart): creates all the instances the bot needs: JNI-BWAPI instance, construction and influence map and assigns values to the variables and lists. Once all this is done, it creates the behavior trees structure. Leaf nodes implementation is developed in different classes.
  - matchFrame (equals to BWAPI's onFrame): executes all the behavior trees in this order: Gathering, Build, Add-ons, Train, Defense, Update

Troops and Attack. After executing all the behavior trees, the influence map is updated if the number of frames is multiple of 300.

- unitShow (equals to BWAPI's onUnitShow): used for detecting enemy race and to select the list of units to train during the match.
- unitCreate (equals to BWAPI's onCreate): every time a building is starting to be built is added to the specific control list.
- unitComplete (equals to BWAPI's onUnitComplete): every time a building or unit is completed the influence map and construction map are updated. If the unit or building's owner is the bot, then it is added to the specific control list.
- unitDestroy **(**equals to BWAPI's onUnitDestroy): every time a unit or building is destroyed is removed from the specific control list and the influence is updated.
- unitMorph (equals to BWAPI's onUnitMorph): due to the internal API issues, it is used to know when a refinery is built.

- **Variables:**
  - Influence map: it is a matrix representation of the match map. Each cell has assigned a numeric value (influence): positive values if it is the player and negative values if it is the enemy or 0 if neutral. A bigger value means a higher player/enemy control of that position. The influence generated by units and buildings is updated each 300 frames.
  - Construction map: matrix with dimension X x Y which are the width and height of the map respectively. Each cell can have a value from 0 to 6, according to the max size of the building which can be built in that position taken the top-left corner as reference.
  - Control variables: group of list and variables which are used to have an internal control of the match. Also they are used to reduce the computing time thanks to the direct access to units or buildings.
- **Behavior Trees:** Execution order and its reason can be seen in the Illustration 74, executing it in up-down, left-right order.

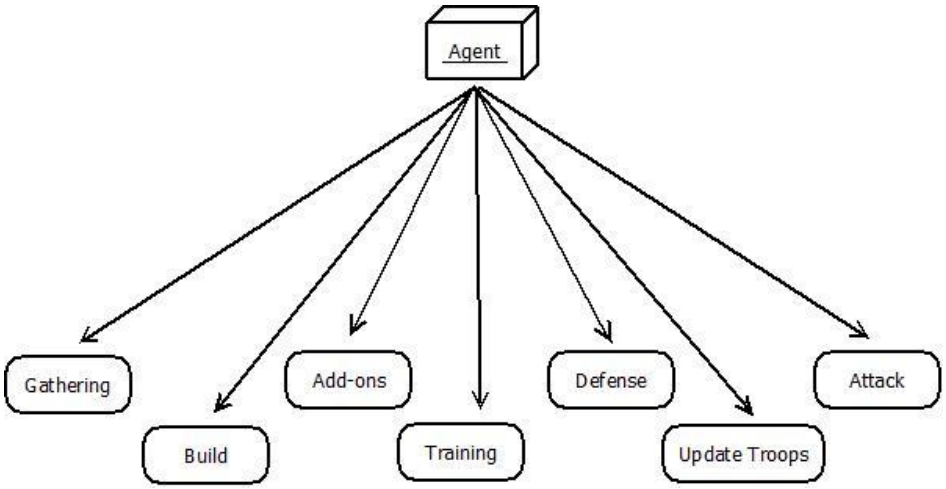


Illustration 74 Vision of agent in tree

- Gathering: executes first because of the importance of gathering resources. Without the gathering of resources the rest of behaviors will not work correctly.
- Build: after gathering resources, this is the second most important; as buildings are used to create new units for attacking and defending.
- Add-ons: for this tree the order does not matter. It builds add-ons for the built buildings.
- Training**:** goes after Build to give the chance to have buildings to train units before Attack and Defense. There is no point in attack or defense without units.
- Defense**:** goes before any action of attacking because it is important to defend correctly the base before attacking.
- Update Troops**:** executes before Attack to optimize the troops control variables (merge, remove…).
- Attack**:** Goes after Update Troops to make sure the existent control variables of troops are correct.

7.3.3. Influence map

The influence map is one of the variables in 7.3.2. Agent. It is the key for attack and defense unit movements.

An influence map is a representation in cells of the game scenario. Each cell has attached an influence value which represents the player or enemy control of the cell.

The influence is represented with numeric values, positive values for the player and his/her allies, and negative values for the enemy. A 0 value indicates a neutral cell. This influence is generated by units and buildings in the game, each one with a specific value. It must be updated during the match because the movement, creation and destroy of it makes the control zones change.

Influence map is used to determinate which zone attack as negative cells are places where the enemy has units or buildings. It is implemented as a bidimensional matrix X x Y, where X is the width and Y is the height of the map.

7.3.4. Gathering

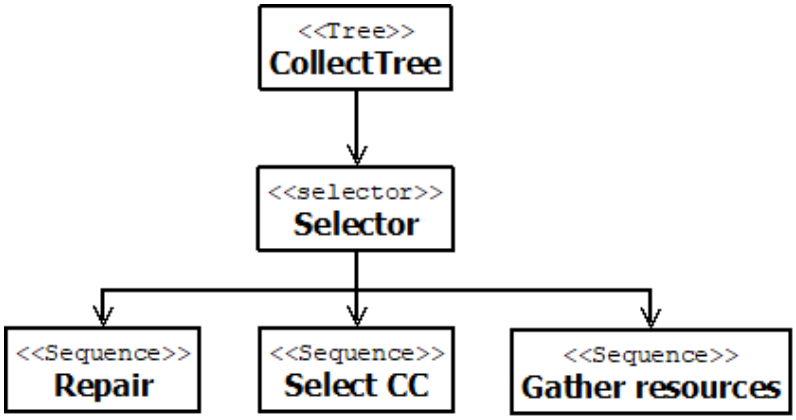


Illustration 75 Behavior tree: Gathering

This behavior tree (Illustration 75) manages actions related with SCVs. It is formed by three sequences: *Repair, Select CC* and *Gather resources.*

- *Repair*: This sequence is responsible of identifying if exists some buildings damaged and if that is the case, finds an idle worker to repair the buildings.
- *Select CC*: This sequence is responsible of selecting a Command Center from all the Command Centers built.
- *Gather resources*: This sequence is responsible of selecting an idle worker and send it to gather resources. The optimal number of workers to gather resources is three, because while a worker is gathering, the other one is delivering and the third one is waiting to start immediately to gather resources when the previous one finishes. There are two possibilities:
    - Gather vespin gas: If exists a refinery but there are not three workers gathering vespin gas, it sends the worker to gather vespin gas and returns true. Otherwise it returns false and starts gathering minerals.
    - Gather minerals: If the number of workers is less than three per mineral node, sends the worker to gather mineral.

7.3.5. Training

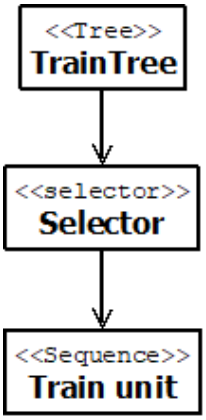


Illustration 76 Behavior tree: Training

This behavior tree (Illustration 76) manages the process to train new units. It is formed by the sequence *Train unit*.

- Train unit: To train a new unit, the sequence follows the next steps:
    1. Firstly checks if the agent has enough resources (mineral and vespin gas) to pay the unit cost. In that case, passes to the next step.
    2. Finds the building which can train the desired unit.
    3. If the building is idle, starts to train the new unit.

7.3.6. Build

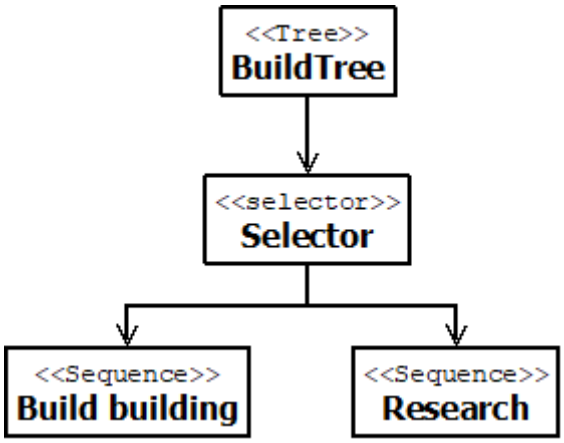


Illustration 77 Behavior tree: Build

This behavior tree (Illustration 77) manages two processes: build new buildings and researchs new technology.

- Build buildings: to build new buildings, it has to perform the following actions:
    1. Check resources: checks if the agent has enough resources (mineral and vespin gas) to pay for the building cost. If that is the case, passes to the second step.

2. Find position: finds a valid position to build. If finds a valid position passes to the next step.
3. Find builder: finds an idle worker to order to build the building. If there is an idle worker, passes to the next step.
4. Move: moves the worker to the found position because to be able to build needs a position without fog of war.
5. Build: orders to the worker to build the building in the found position. Returns true if can build, false otherwise.

- Research new tech: in the case that there is no need or cannot build a new building. The steps to research new technology are:
    1. Check resources: checks if the agent has enough resources (mineral and vespin gas) to pay for the research.
    2. Research: in case that there are enough resources, finds the building which can do the research and starts it.

7.3.7. Add-ons

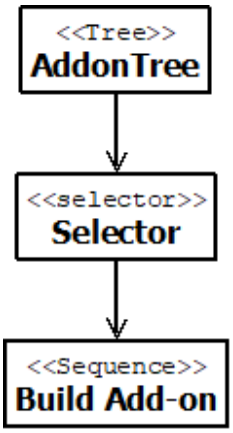


Illustration 78 Behavior tree: Add-ons

This behavior tree (Illustration 78) manages the process of building add-ons on specific buildings. The steps to build the add-on of certain building are:

1. Check resources: Checks if the agent has enough resources (mineral and vespin gas) to pay for the add-on.
2. Find building: Finds a building without the add-on.
3. Build: If there is a building without the add-on, builds it.

7.3.8. Attack

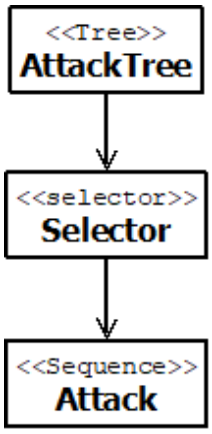


Illustration 79 Behavior tree: Attack

This behavior tree (Illustration 79) manages the attack. To have an easier way to manage it, a class *Troop* has been implemented.

The steps to successfully attack are:

1. Check State: Checks troops states, removing empty troops from the control list and updating its attributes.
2. Choose destination: Finds a valid position to send the troop.
3. Select troop: Selects an inactive troop to give orders.
4. Order: If the units of the troop are close to each it others order to attack to the position found. Otherwise, it orders them to regroup going to a closer position.

7.3.9. Update Troops

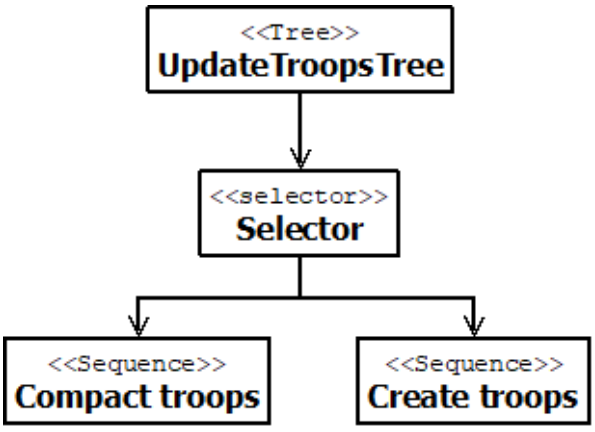


Illustration 80 Behavior tree: Update Troops

This behavior tree (Illustration 80) is responsible of keeping the troops updated. To make this, it has two sequences:

- Compact troops: Removes empty troops from the control lists, checks states and reorders the units to avoid units with less than ten units.

- Create troops: If all troops are full (ten or more units) and there are units with no soldiers, it creates a new empty troop.

7.3.10. Defense

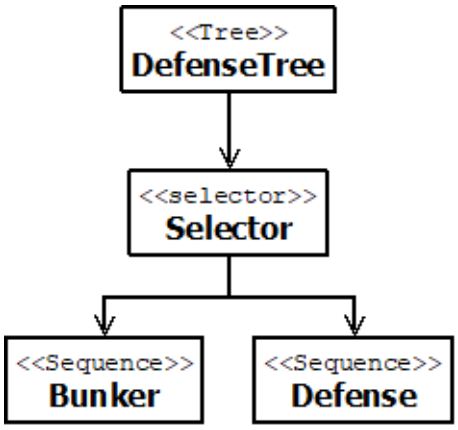


Illustration 81 Behavior tree: Defense

This behavior tree (Illustration 81) manages the two main defense behaviors:

- Bunker: This sequence checks for bunkers not filled. In that case, it orders to idle marines to go and load the bunker.
- Defense: When all bunkers are filled it orders to the military units to move to a defensive position. The defensive position is defined at the beginning of the match. There are two possibilities:
  - If there is only one choke point as entry to the base that will be the defensive position.
  - If there is more than one choke point, the defensive position is the ground near to the Command Center.

7.4. Experimentation

In this section is described the experimentation carried out for the initial and last version of the bot, observing how each version improves respecting to its previous version.

7.4.1. Used maps

The maps used have been four: Astral Balance, Fire Walker, Breaking Point and Full Circle. A view of each map can be seen in the Illustration 82 Minimap view.

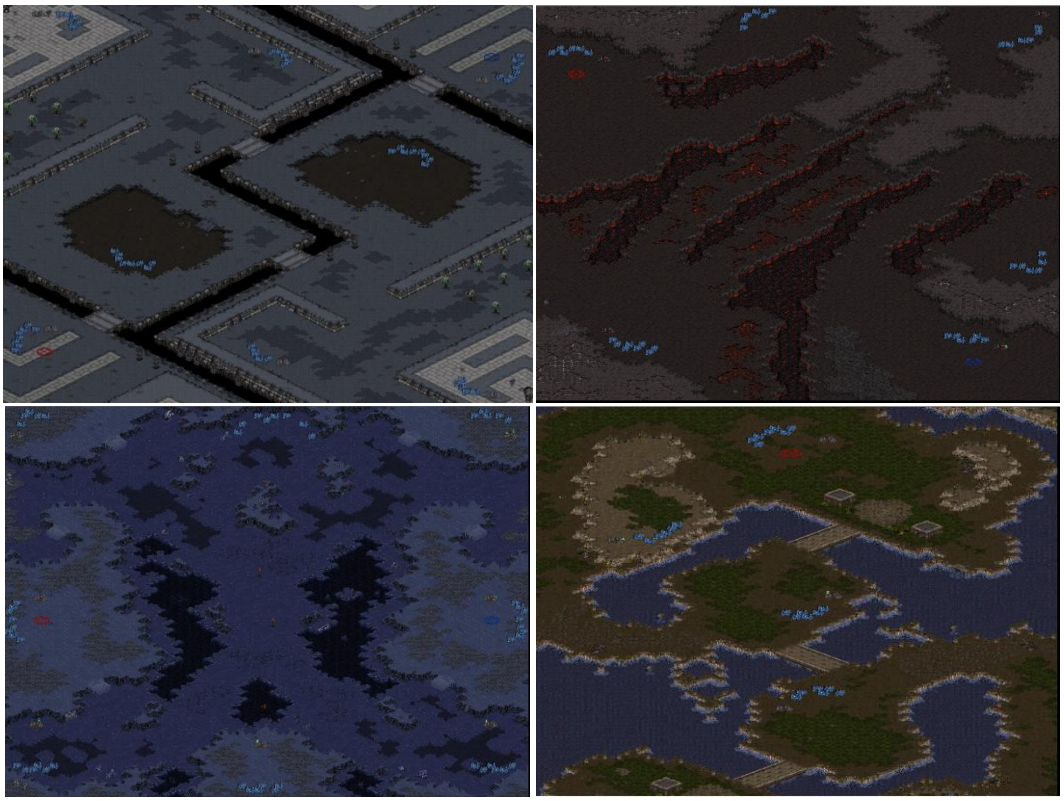


Illustration 82 Minimap view

The reasons for choosing each map are:

- **Astral Balance**: Map with only one entry to the initial base and multiple expansions available which are more or less protected (one entry and/or different ground level). It has choke points and various ground levels, including zones only accessible to air units.
- **Fire Walker**: Similar to Astral Balance, but without inaccessible zones. It has only one entry to the base and close expansion.
- **Breaking Point**: Map with multiple expansions and various entries to the base. The map structure only allow go to two different path to attack the enemy, which forces the units to go through the expansions.
- **Full Circle**: Map with multiple expansions and various entries to the base. The entire map is accessible with ground units. There is one main path which passes through almost all expansions.

7.4.2. Influence evolution

Influence map is the main technique used to control attack and defense, but also the construction of certain buildings (few buildings) during the match. Without it, the bot will neither win nor hold the enemy's attacks.

Due to its importance and use, among the different versions, the values assigned to each unit have been changed and updated.

7.4.2.1. Version 0 influence

In this first version the following criteria for update the map influence is applied:

- Updated each 300 frames.
- Applies following values for building:
  - Attack capable buildings: applies influence of 4 to the map. Includes all buildings which supports "*isAttackCapable()*".
  - Defensive buildings: applies influence of 5 to the map. Includes bunkers (*Terran*), Missile Turrets (*Terran*) and Spore Colony (*Zerg*)
  - Others: applies influence of 3 to the map. Includes the rest of the building.
- Applies next values for units:
  - Mechanic: applies influence of 3 to the map. Includes every ground vehicle.
  - Air: applies influence of 6 to the map. Includes every air vehicle.
  - Others: applies influence of 1 to the map. Includes the rest of the units.
- The influence for each unit updates only when this unit has moved. If so, the new influence is applied and the old one is removed.

The most problematic aspect of this version is the updating influence process. As it can be seen in the experiments (link) carried out, creating groups of unit do not change its position because they are in negative influence and as they do not move, the influence does not update and never disappears.

7.4.2.2. Version 1 influence

In this version all values and methods do not change as this version is focused in building and influence is not relevant.

7.4.2.3. Version 2 influence

In this version the following improvement has been made:

- Influence values do not change, but now influence is only applied to units that are not workers because its influence over the map is not relevant and only adds noise.

This minor change partially fixes a problem which can be easily seen in the version 0 experiments, now the troops will stay less time locked in the enemy minerals because now the workers' influence is not considered on the map.

7.4.2.4. Version 3 influence

In this version significant changes have been made in application and updating, which almost removes the blockage of the troops' movement.

- Separation of the buildings and units' influence update, because must be treated differently. Building should only affect to the map influence once they are completed, therefore it will stay constant from that moment. Because units can move, its influence must be updated during the match, removing the previous and applying the new one.
- New influences for buildings:
    - Bases: applies influence of 7 to the map. Includes each race's bases and applies the maximum influence because they are very relevant buildings.
- New influence for units:
    - Others: Applies influence of 2 instead of 1 to the map.
    - Mechanic and Air units do not change.
- New influence update: now the influence to update is the unit's influence, not buildings', and it will be always updated. The process consists on: remove previous influence and apply unit's influence into its current position.

This new update and values for buildings and units almost removes the blockage of movement as it can be seen on the experiments carried out (link).

7.4.2.5. Version 4 influence

This version includes minor fixes and new influences for units and buildings.

- New influence for buildings: new criteria and order:
    - Important buildings: applies influence of 5 to the map. That includes buildings like: *Pylons* (*Protoss*), *Bunkers* (*Terran*), *Spore Colony* and *Sunken Colony* (*Zerg*) are buildings to consider, because its importance (*Pylons*) or because indicate a well defended areas (*Bunker* or *Colonies*).
    - Attack capable buildings: applies influence of 4. It includes all buildings which support "*isAttackCapable()*".
    - Bases: do not change, have the same influence value (7).
    - Others: do not change, have the same influence value (3).
- New influence for units: Every influence has changed, reducing values.
    - Mechanic: Value changed from 3 to 2.
    - Air: Value changed from 6 to 3.
    - Others: Value changed from 2 to 1.

The most significant improvement is the new order to apply the influence, as in previous versions it was being applied in wrong order. For instance, *Spore Colony* is considered a defensive building (influence of 5), but it can also attack and with the first condition "*isAttackCapable()*" made that it was having an influence of 4 instead of 5. Additionally it has been added two important buildings: *pylon*, because it is the spine of *Protoss* buildings and *Sunken Colony* which is other important defensive *Zerg* building.

The reduction of the units' influence is needed because it was "obfuscating" the values of the buildings, which are the real final objective. This reduction allows the agent have a better visualization of the enemy bases.

7.4.3. Experimentation vs Game AI

This section contains all the experimentation executed for each version. The following features are going to be evaluated:

- **Resources**:
  - Mineral: It corresponds to all mineral gathered by the agent during the match.
  - Gas: It corresponds to all vespin gas gathered by the agent during the match
  - Spent: It corresponds to all resources (mineral and vespin gas) spent by the agent during the match.
- **Units**:
  - Trained: It corresponds to all ally units trained during the match.
  - Lost: It corresponds to all ally units killed by the enemy during the match.
  - Eliminated: It corresponds to all enemy units killed by ally units during the match.
- **Buildings**:
  - Buildings: It corresponds to all ally buildings built during the match.
  - Lost: It corresponds to all ally buildings destroyed by the enemy during the match.
  - Eliminated: It corresponds to all enemy buildings destroyed by ally units during the match.

The figures shown below for each version have been created with the data collected after play, at least, two matches against each race on each map indicated in 7.4.1. Used maps. Calculating the mean of each feature.

Feature / Version	Build	Train basic units	Attack	Defense	Manage expansions	Detect clocked units	Train advance units	Research tech
Version 0	✔	✔	✔	✔	✔	✔	✔	✔
Version 1	✔	✔	✔	✔	✔	✔	✔	✔
Version 2	✔	✔	✔	✔	✔	✔	✔	✔
Version 3	✔	✔	✔	✔	✔	✔	✔	✔
Versión 4	✔	✔	✔	✔	✔	✔	✔	✔

Table 42 Version feature summary

In Table 42 Version feature summary can be seen with the following criteria the different features of each version:

- Green with tick: Features implemented working at 100%.
- Yellow with tick: Features implemented but have errors or needs more work.
- Red with cross: Features not implemented.

Data collected can be found in 8.3. Resultados de las pruebas

7.4.3.1. Version 0

Initial version, it is the agent developed in Artificial Intelligent in the Entertainment Industry. It can be considered a baseline agent, as so in the experiment the three features will be evaluated.

All experiments can be found in this link.

Resources

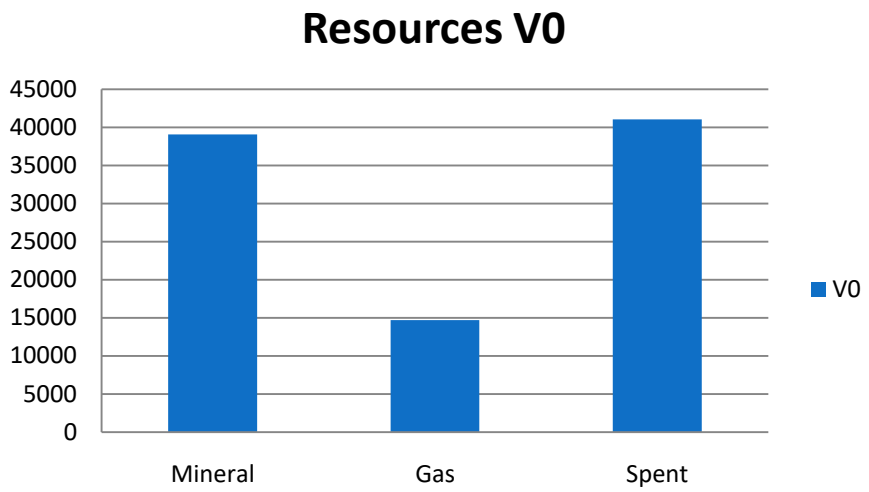


Illustration 83 Resources Version 0

Illustration 83 shows the resources score obtained for version 0. These values are taken as reference for next versions. It is hard to extract a conclusion only from this score.

Units

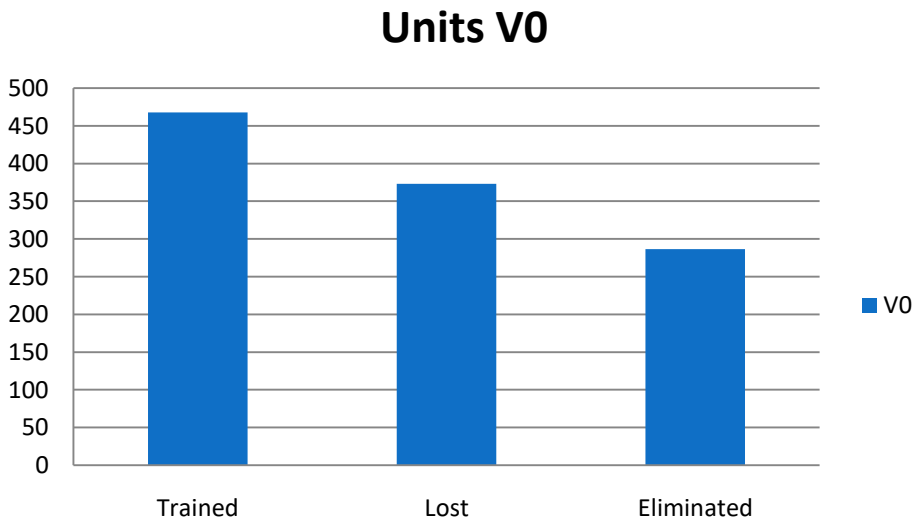


Illustration 84 Units Version 0

Illustration 84 shows the units score obtained for version 0. Unlike the resources one, through the game units we can have a global vision about how the bot plays. An important remark is that unit lost are less than units trained which means that the **bot is capable of winning some matches**, even though its attack and defense behaviors are very rudimentary. On the other hand, having more lost units than eliminated means that **the bot loses more matches than it wins**.

Buildings

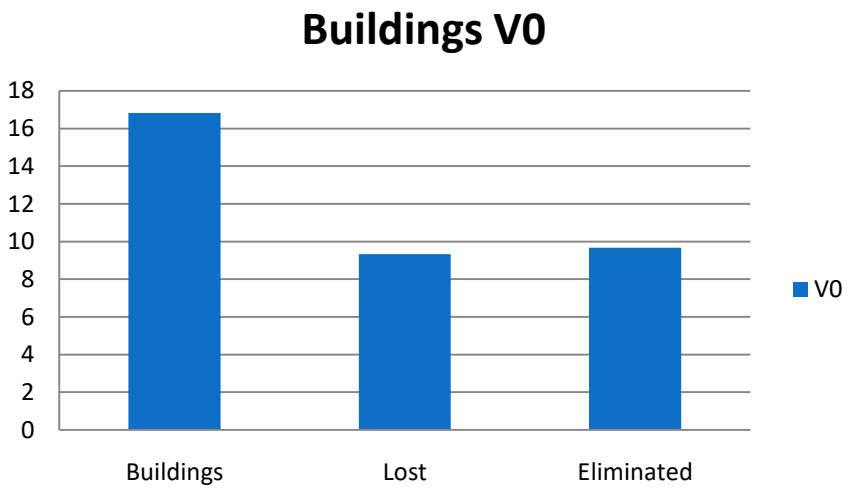


Illustration 85 Buildings Version 0

Illustration 85 shows the building score obtained for version 0. These values are in line with units' values. It can be seen that the bot has an average lose of half of the buildings. If an observation is made to the section in 8.3. Resultados de las pruebas, it can be notice that when the bot loose most of its building it means the bot is playing in maps with multiple entries to the base. This happens because it does not have included a specific behavior for this situations.

As this chapter is a summary of the experiments carried out, it will only contains the version 0 and version 4 experiments. To see the experiments carried out for version 1, 2 and 3 go to the Spanish section of the document.

7.4.3.2. Version 4

This final version has in consideration the enemy race. According to the enemy race, it gives priority to different elements.

All experiments can be found in this link.

Resources

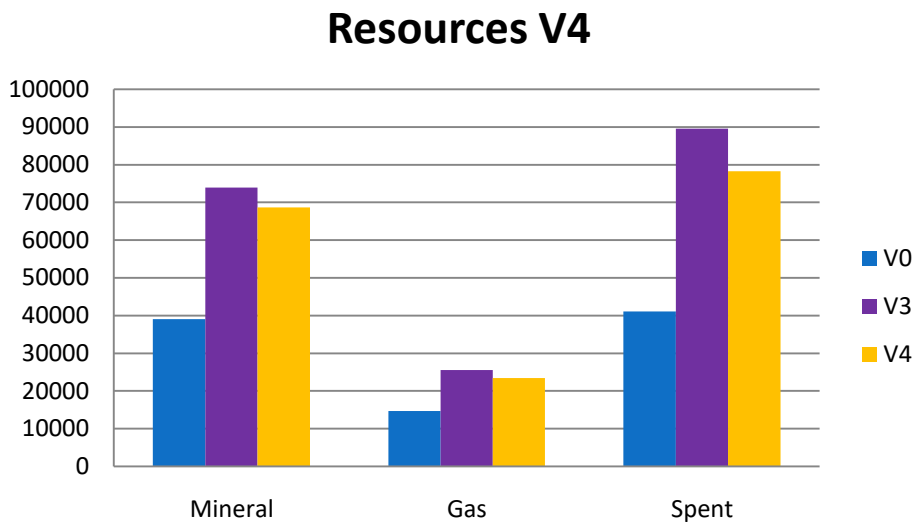


Illustration 86 Resources Version 4

Ilustración 64 shows the units score obtained for version 3 and version 4. Resources do not change with respect to previous versions because it has not have any significant improvement since version 3. The small decrease with respect to version 3 it is because shorter matches.

This increase with respect from version 0 has two main reasons: longer matches and management of expansions. Expansions allow collecting a larger amount of resources.

Units

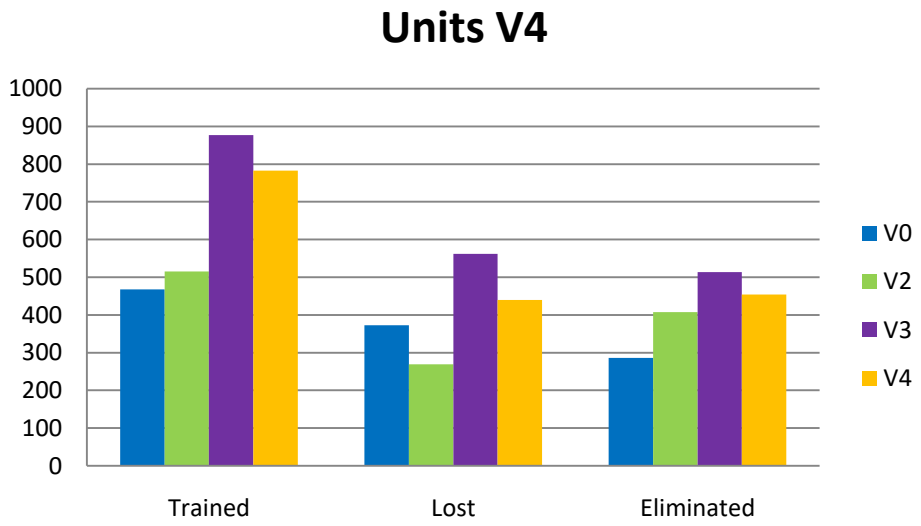


Illustration 87 Units Version 4

Ilustración 65 shows the units score obtained for version 0, version 2, version 3 and version 4. Same as resources, matches are a bit shorter so fewer units are

trained. The significant change occurs in version 2, with real attack and defense management. Then it is again increased with version 3 when the resources gathered increased significantly, allowing to train a larger number of units.

Buildings

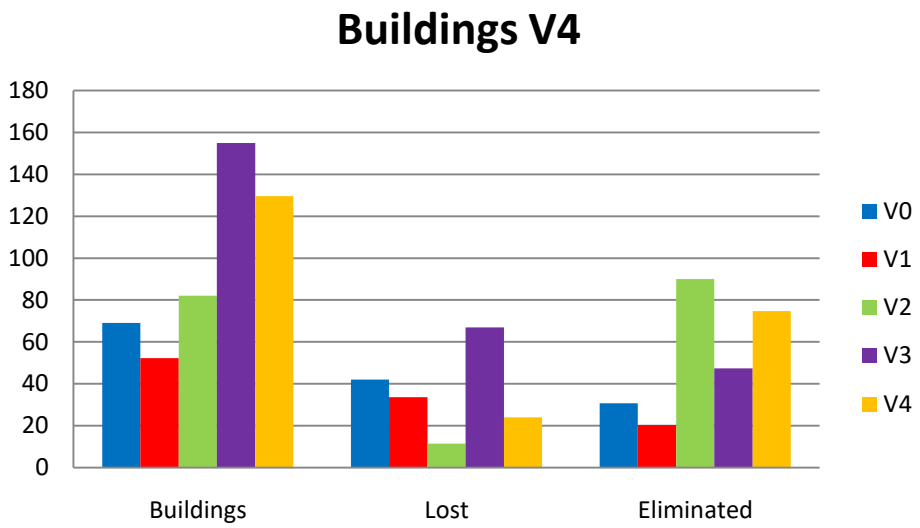


Illustration 88 Buildings Version 4

Illustration 88 shows the buildings score obtained for version 0, version 1, version 2, version 3 and version 4. Buildings suffer a significant change in lost and eliminated buildings. This it is straight consequence of the changes made in the influence map. Thanks to this feature, the bot can have better recognition of enemy position and make more effective attacks. Another straight consequence of this issue are the shorter matches, because the attacks are stronger and blockage is almost removed, which means that the bot wins sooner.

7.5. Conclusions and Future Work

In this chapter it will be first discussed the general conclusions of this work. It will also be mentioned the problems found and what has contributed to me personally. Next a concrete conclusion for each main objective is made and finally, improvements are proposed as future directions of the current work..

7.5.1. Global conclusions

Videogames AI development is quite easy and fast when defining simple behaviors. When the behavior starts to be more complex it is a difficult task but the technique chosen can help to make it easier. Most of the time spent on testing the new behaviors and verifying that they do not enter in conflict with previous ones. Also it forces to make the most efficient code as the answers must be taken in real time or in a very short window of time, which makes the quality-cost balance very important.

Making this work has allowed me to continue the bot developed in "Artificial Intelligent in the Entertainment Industry", going deeper in concrete features and learning more about BWAPI. As a result I have become interested in participating in SSCAIT. Maybe I will also seeing the possibilities maybe I will try to make an AI for other games like 0 A.D. or OpenRA. All of this serves me as first experience in videogame AI development, a field very demanded nowadays.

About problems found, the most important are:

- Detect attacks: While the bot can detect when an enemy is shown, it cannot know when a unit is being attacked or started to attacks, which means that to be able to know this information it must check units' status on each frame. This problem was solved by counting the number of alive units in troops on each frame.
- Buildings: This feature still does not work as it should be, leaving empty space without apparently reason. Most of the time spent in the first versions was used in fixing and testing this feature.
- Movement: The influence map is very useful to obtain which position to attack, but if you want to move the units to positions not identify for the game, it is quite complicated. This problem was especially involved in grouping units when they got separated too much. To fix it, instead of moving to a specific position in the map, they were moved to a closer position where another unit of the troop was placed.

7.5.2. Conclusions concerning objectives

Next a conclusion on whether the objective has been achieved successfully or not and how it is explained as follows:

- Environmental Analysis: This objective has been achieved successfully.
- Choose strategies: This objective has been partially achieved. In last version has been established a specific unit list to train according to the enemy that the bot is facing and whether the enemy trains invisible units. Then it is a priority train a detector. However, it is still pending to define more possible strategies.
- Management attack and defense: This objective has been achieved successfully. Thanks to the implementation of "Troop" with different states, it is possible have a fully management of military units, choosing where to move and actions to do. Although, it can still be improved to have more complex behaviors.

7.5.3. Future Work

The bot developed is not completely finished, as some work needs to be done still. Most of this work consists on adding more techniques and behaviors which can be necessary to guarantee the victory.

As for the techniques, the most important and urgent are:

- Establish measures to avoid moving blockage of movement through choke points. This could be done by having a timer which counts the time a unit is idle during the match, or if the regrouping state does not change in a determinate time.
- Capability of load and unload unit to access inaccessible high ground terrain. A good way to start defining these behaviors is using a specific tree for "extra" behaviors such as: Using abilities or move specific units as dropships.
- Make a micro management for attacking. Units as vultures are most effective when they are micro managed. The usual way to do this is locking an enemy first and only attack him when it is in the limit of the attack range. If the enemy comes closer then run away until the enemy it is in the limit of the attack range.

In attack and defense behaviors, an important feature which could be implemented is the analysis of closer enemies, doing that it could predict if it will win or lose the fight or focus the attack on specific units.

Finally another important aspect is change from JNI-BWAPI to BWMirror as long as we want to continue developing with Java as language. A different idea will be to translate the bot to C++ to use BWAPI. Both options will allow us to use a better updated version and take advantage of bug fixes and efficiency improvements.

Capítulo 8: Anexos

8.1. Manual de instalación

Para la instalación y uso del proyecto desarrollado se necesitan descargar e instalar los siguientes programas:

- Java JDK 8 de 32 bits: Se puede encontrar en este enlace: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
- Poseer una copia física de *StarCraft* y *StarCraft: Brood War*. Una vez instalado, actualizar a la versión 1.16.1 que se puede encontrar en el siguiente enlace (Ilustración 89): https://us.battle.net/support/en/article/*StarCraft*-patch-information

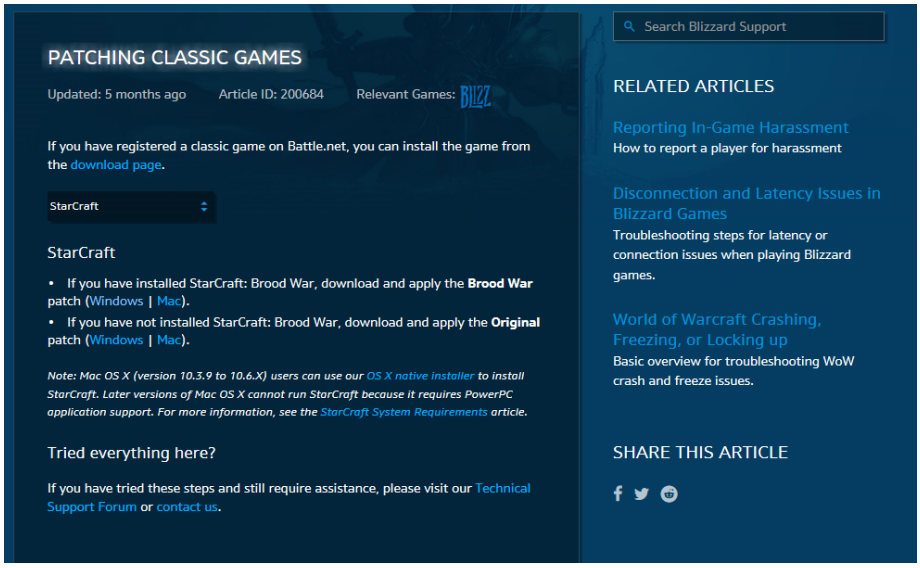


Ilustración 89 Descarga del parche 1.16.1

Para ello seleccionamos *StarCraft* como juego y nos descargamos la versión para Windows o Mac, según corresponda.

- BWAPI 3.7.5, que se puede encontrar en el siguiente enlace: https://github.com/bwapi/bwapi/releases
    - Si BWAPI no incluyera ChaosLauncher por cualquier motivo, puede encontrarse en el siguiente enlace (Ilustración 90): http://www.teamliquid.net/forum/brood-war/65196-chaoslauncher-for-1161

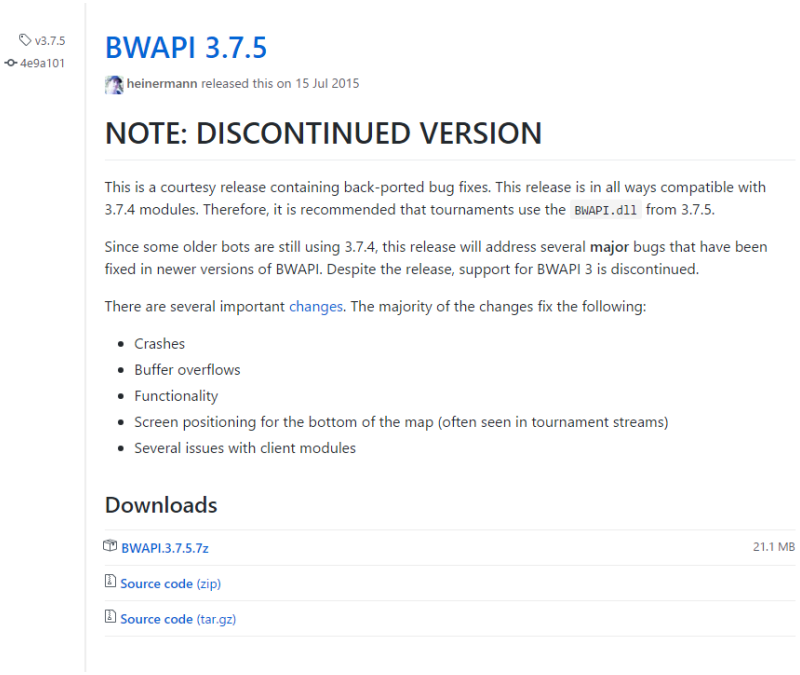


Ilustración 90 Descarga de BWAPI

- Ejecutable del agente, que se puede encontrar en el siguiente enlace:

Una vez tenemos todo lo necesario descargado podemos proceder a la instalación de los mismos en el siguiente orden:

1. Instalar Java JDK 8, basta con ejecutar el ejecutable que nos hemos descargado.
2. Instalar *StarCraft: Brood War* y actualizar ejecutando el ejecutable que nos hemos descargado previamente.
3. Una vez instalado Java y *StarCraft*, tenemos que instalar BWAPI. Para ello hay que copiar el contenido de dentro del 7zip a la carpeta raíz de nuestro ordenador, en caso de Windows, a C: o el disco correspondiente.

   Una vez ahí, copiamos el contenido de la carpeta Windows de BWAPI en la carpeta Windows de nuestro sistema y la carpeta *StarCraft* en la carpeta donde *StarCraft* se haya instalado. Para verificar que funciona correctamente ejecutamos el ejecutable "*ChaosLauncher*.exe" presente en la carpeta "BWAPI 3.7.5/*ChaosLauncher*", si todo funciona correctamente se nos abrirá (puede que salga un error y una advertencia, los ignoramos) y deberemos activar, si no está activado, el inyector de código y el modo ventana (por comodidad) (Ilustración 91).

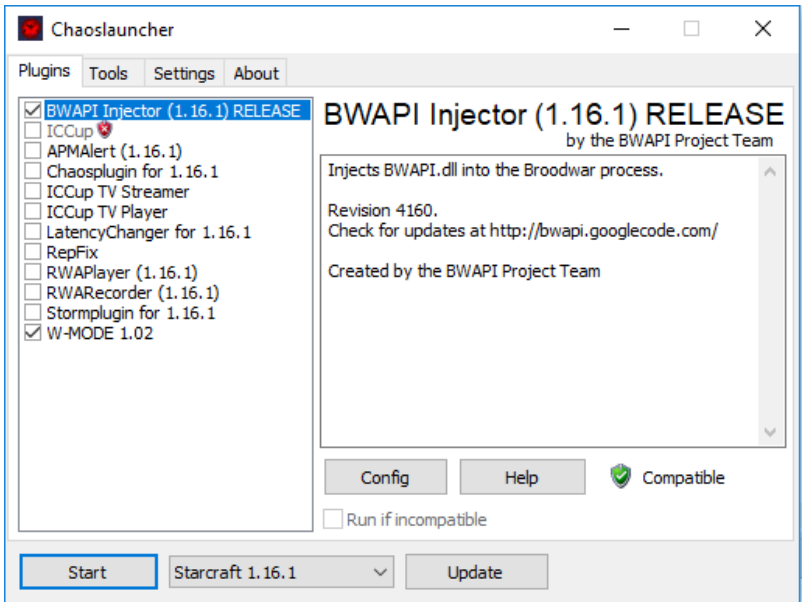


Ilustración 91 Ventana Plugins de ChaosLauncher

Pulsamos el botón "Config" y se nos abrirá un editor de texto, cambiamos el valor de "ai" y "ai_dbg" a NULL y guardamos los cambios realizados. Se encuentran al comienzo del fichero (Ilustración 92).

```
[ai]
; Paths and revisions for AI
;   - Use commas to specify AI for multiple instances.
;   - If there are more instances than the amount of
;         DLLs specified, then the last entry is used.
;   - Use a colon to forcefully load the revision specified.
;   - Example: SomeAI.dll:3400, SecondInstance.dll, ThirdInstance.dll
ai     = NULL
ai_dbg = NULL
; Used only for tournaments
```

Ilustración 92 Cambios bwapi.ini

Si todo va bien y pulsamos "Start" debería iniciarse el juego en modo ventana. Si falla y nos indica que no encuentra *StarCraft*, deberemos revisar la pestaña "Settings" y donde dice "Installpath" indicar la ruta donde tenemos instalado *StarCraft* (Ilustración 93).

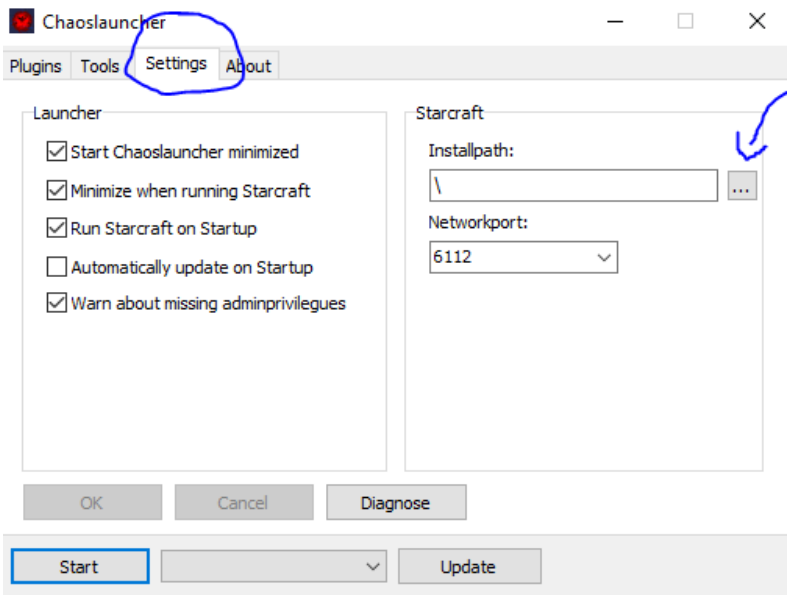


Ilustración 93 ChaosLauncher pestaña Settings

4. Ahora ya podemos ejecutar nuestro agente, para ello:
   4.1. Descomprimimos el contenido de "Agente.zip" donde queramos, en este caso en la carpeta Descargas.
   4.2. Abrimos una "Consola de comandos" o "Símbolo del sistema". En la consola de comandos escribimos dependiendo de nuestro Sistema Operativo y donde hayamos descomprimido la carpeta:

   `"cd Downloads\Agente" ó "cd Descargas\Agente"`

   4.3. Una vez en la carpeta, para ejecutar el agente hay que escribir:

   `"java –jar goliat.jar -a bot.Goliat -s X -u -d -i"`

   Siendo X la velocidad a la que queremos que se ejecute el juego, entre 0 y 10.
5. Si todo va bien, nos saldrá este mensaje en la consola, que indica que está esperando a que se inicie partida (Ilustración 94):

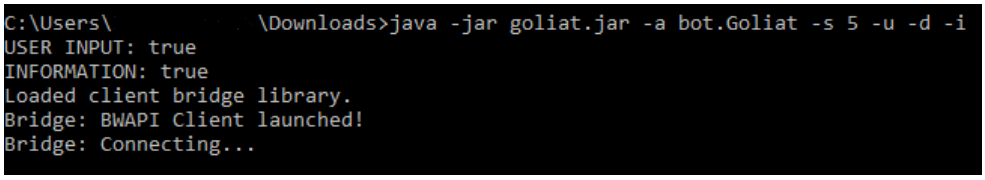


Ilustración 94 Mensaje al ejecutar el agente

6. Ahora hay que abrir el juego, seleccionamos "Un solo jugador" →"Expansión" → "Otros escenarios" (Ilustración 95).

Ilustración 95 Menú StarCraft

7. Aquí ya seleccionamos cualquier mapa y **como raza de nuestro jugador debemos seleccionar Terran**, esto es importante porque si no, el agente no funciona, y listo. El agente jugará la partida por nosotros.
   7.1. **NOTA:** La primera vez que se ejecute en cada mapa tardará un poco en comenzar a funcionar debido a que debe procesar y crear toda la información del mapa en cuestión.

8.2. Manual de usuario

En este apartado se realiza una descripción visual sobre cómo utilizar el agente desarrollado suponiendo que se tienen descargadas e instaladas todas las dependencias. Todas los archivos de las pruebas se pueden encontrar en el siguiente enlace o si se prefieren los videos en este enlace.

1. Abrir *ChaosLauncher* e iniciar *StarCraft* con el inyector de código habilitado (Ilustración 96).

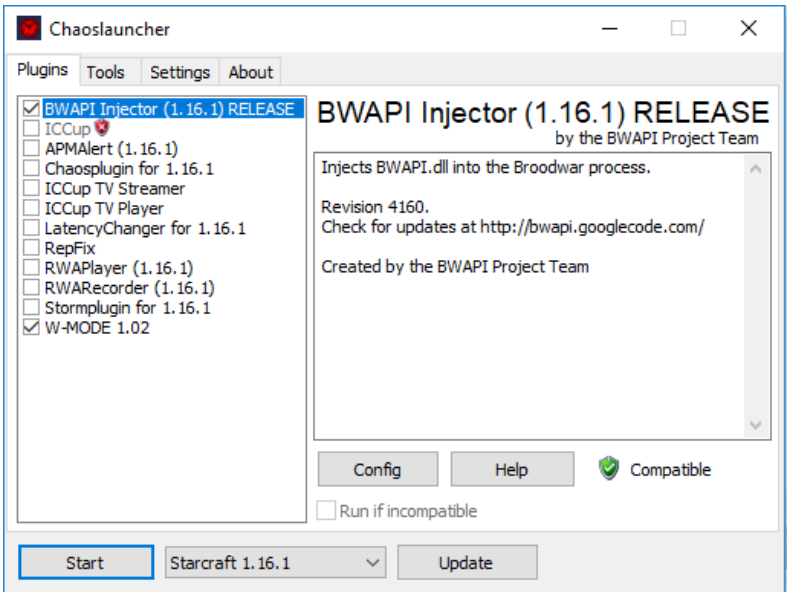


Ilustración 96 Ventana ChaosLauncher

2. Abrir una "Consola de comandos" o "Símbolo del sistema". En la consola de comandos nos movemos a la carpeta donde hayamos descomprimido el agente:

   "cd <Ruta a la carpeta del Agente>"

3. Una vez en la carpeta, para ejecutar el agente hay que escribir:

   "java –jar goliat.jar -a bot.Goliat -s X -u -d -i"

   Siendo X la velocidad a la que queremos que se ejecute el juego, entre 0 y 10.

4. Si todo va bien, nos saldrá este mensaje en la consola, que indica que está esperando a que se inicie partida (Ilustración 97):

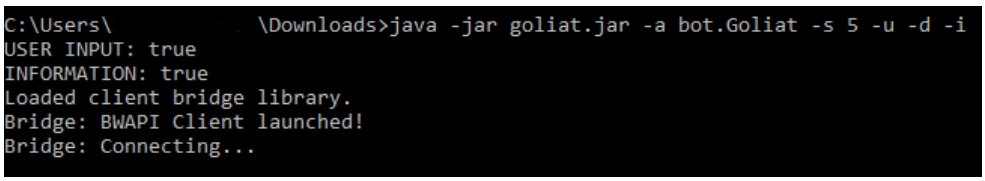


Ilustración 97 Mensaje al ejecutar el agente (2)

5. Ahora hay que abrir el juego, seleccionamos "Un solo jugador" →"Expansión" → Seleccionamos o creamos un perfil -> "Otros escenarios" (Ilustración 98).

Ilustración 98 Menú StarCraft (2)

6. Aquí ya seleccionamos cualquier mapa y **como raza de nuestro jugador debemos seleccionar Terran**, esto es importante porque si no, el agente no funciona, y listo. El agente jugará la partida por nosotros.
    a. **NOTA:** La primera vez que se ejecute en cada mapa tardará un poco en comenzar a funcionar debido a que debe procesar y crear toda la información del mapa en cuestión.

8.3. Resultados de las pruebas

En este apartado se incluyen todos los datos extraídos a raíz de la experimentación realizada (Ilustración 99). Y que se pueden encontrar en el siguiente enlace.

Astral Balance													
	Recursos			Edificios				Unidades					
	Mineral	Gas	Gastado	Total	Construidos	Perdidos	Eliminados	Total	Entrenadas	Perdidas	Eliminadas	Total	
V0	11308,5	4267,66667	11979,3333	15576,3333	16,8333333	9,33333333	9,66666667	5755,83333	139,333333	113	106,166667	32051,6667	53383,8333
V1	-	-	-	-	12,6666667	12,6666667	2	1901,66667	-	-	-	-	
V2	-	-	-	-	19,5	0,5	27,5	10240	141	63,5	122	33250	
V3	18950	6180,66667	22525	25130,6667	39,3333333	21,3333333	12,3333333	8470	233,333333	128	166,333333	46565	80165,6667
V4	17721,3333	3875,66667	18070,8333	21597	33,1666667	0,5	22,3333333	11029,1667	184	64,1666667	110	34790,8333	67417
Fire Walker													
	Recursos			Edificios				Unidades					
	Mineral	Gas	Gastado	Total	Construidos	Perdidos	Eliminados	Total	Entrenadas	Perdidas	Eliminadas	Total	
V0	7943,33333	3244,66667	8201,66667	11178,6667	14,3333333	5	11,6666667	5138,33333	93,6666667	66	61,6666667	16291,6667	32608,6667
V1	-	-	-	-	19	0	18,3333333	7306,66667	-	-	-	-	
V2	-	-	-	-	22	0,5	23,5	10052,5	114,5	46	91,5	22862,5	
V3	15654	4994	20066,6667	20648	36,3333333	0,33333333	15	8921,66667	190,666667	78	98	37216,6667	66786,3333
V4	11698,6667	3520,66667	14050	15219,3333	25,5	1,66666667	13,1666667	6589,16667	144,333333	66	86,1666667	24173,3333	45981,8333
Full Circle													
	Recursos			Edificios				Unidades					
	Mineral	Gas	Gastado	Total	Construidos	Perdidos	Eliminados	Total	Entrenadas	Perdidas	Eliminadas	Total	
V0	9256,66667	3250,33333	9729,66667	12507	19,5	13,1666667	5	3353,33333	109,333333	81,3333333	54,5	21430	37290,3333
V1	-	-	-	-	10,3333333	10,6666667	0	1196,66667	-	-	-	-	
V2	-	-	-	-	23	1	30,5	11527,5	144,5	81,5	137	32862,5	
V3	22723,3333	7984	28383,3333	27374	48	14	17,6666667	11293,3333	257,333333	160,666667	173,666667	65026,6667	103694
V4	20184,6667	7723,66667	24287,5	27908,3333	37,5	5,5	24,6666667	11387,5	241,166667	159,333333	141,833333	51987,5	91283,3333
Breaking Point													
	Recursos			Edificios				Unidades					
	Mineral	Gas	Gastado	Total	Construidos	Perdidos	Eliminados	Total	Entrenadas	Perdidas	Eliminadas	Total	
V0	10558	3948,33333	11150,1667	14506,3333	18,3333333	14,5	4,33333333	3040,83333	125,333333	112,666667	64	20970	38517,1667
V1	-	-	-	-	10,3333333	10,3333333	0	1030	-	-	-	-	
V2	-	-	-	-	17,5	9,5	8,5	3915	115,5	78	57	12825	
V3	16595,3333	6417,33333	18592,3333	23009,3333	31,3333333	31,3333333	2,33333333	3816,66667	195,333333	195,333333	76	36100	62926
V4	19056,6667	8276,5	21845,8333	27333,3333	33,3333333	16,3333333	14,5	8230	212,833333	150	115,833333	44890	80453,3333
TOTAL													
	Recursos			Edificios				Unidades					
	Mineral	Gas	Gastado	Total	Construidos	Perdidos	Eliminados	Total	Entrenadas	Perdidas	Eliminadas	Total	
V0	39066,5	14711	41060,8333	53768,3333	69	42	30,6666667	17288,3333	467,666667	373	286,333333	90743,3333	161800
V1	#¡VALOR!	#¡VALOR!	#¡VALOR!	#¡VALOR!	52,3333333	33,6666667	20,3333333	11435	#¡VALOR!	#¡VALOR!	#¡VALOR!	#¡VALOR!	0
V2	#¡VALOR!	#¡VALOR!	#¡VALOR!	#¡VALOR!	82	11,5	90	35735	515,5	269	407,5	101800	0
V3	73922,6667	25576	89567,3333	96162	155	67	47,3333333	32501,6667	876,666667	562	514	184908,333	313572
V4	68661,3333	23396,5	78254,1667	92058	129,5	24	74,6666667	37235,8333	782,333333	439,5	453,833333	155841,667	285135,5

Ilustración 99 Resultados pruebas

Referencias

[999 Games, 2003] BoardGameGeek, *¡Pingüinos! - Hey, That's My Fish*. https://boardgamegeek.com/boardgame/8203/hey-s-my-fish [2017, 14/06]

[AEV, 2017] Asociación Española de Videojuegos 2017, *¿Sabías qué? La industria del Videojuego*. Disponible en: http://www.aevi.org.es/la-industria-del-videojuego/sabias-que/ [2017, 05/01].

[AIIDE, 2017] *AIIDE Starcraft AI Competition*. Disponible en: http://www.cs.mun.ca/~dchurchill/*StarCraft*aicomp/ [2017, 16/05].

[Blizzard, 1994] Blizzard Entertainment, página de Warcraft: Orcs & Humans. Disponible en: http://us.blizzard.com/es-mx/games/legacy/ [2017, 13/06].

[Blizzard, 1998a] Blizzard Entertainment página official de *StarCraft*. Disponible en: http://eu.blizzard.com/es-es/games/sc/ [2017, 05/01]

[Blizzard, 1998b] Blizzard Entertainment, *FAQ - AI Scripts*. Disponible en: http://classic.battle.net/scc/faq/aiscripts.shtml [2017, 05/01].

[BWAPI, 2017] *BWAPI*, repositorio oficial. Disponible en: https://github.com/bwapi/bwapi [2017, 19/05].

[Churchill y Buro, 2011] Churchill, D. & Buro, M. 2011, *Build Order Optimization in StarCraft*, *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital EntertainmentAAAI Press*, pp. 14.

[Corkill y Daniel, 1997] Corkill, Daniel D. *Countdown to success: Dynamic objects, GBB, and RADARSAT-1*. Communications of the ACM 40.5 (1997): 48-58.

[Ensemble Studios, 1997] Ensemble Studios, página official de Age of Empires. Disponible en: https://www.ageofempires.com/ [2017, 13/06].

[eSports prizes] eSports earnings, lista de premios de torneos. Disponible en: https://www.esportsearnings.com/tournaments [16/06]

[Games Workshop, 1983] https://www.games-workshop.com/Warhammer

[Gobierno de España, 1996] Gobierno de España, 22/04/1996, *Real Decreto Legislativo 1/1996*. Disponible en: https://www.boe.es/buscar/act.php?id=BOE-A-1996-8930.

[González, 2011] González Morcillo C., Lógica *Difusa, Una introducción práctica*. Disponible en: http://www.esi.uclm.es/www/cglez/downloads/docencia/2011_Softcomputing/LogicaDifusa.pdf [2017, 15/06].

[Google, 2016] *DeepMind and Blizzard to release StarCraft II as an AI research environment*. Disponible en: https://deepmind.com/blog/deepmind-and-blizzard-release-*StarCraft*-ii-ai-research-environment/ [2017, 16/05].

[Haemimont games, 1997] Haemimont Games, páginas official de los creadores de Tzar. Disponible en: https://www.haemimontgames.com/ [2017, 13/06].

[IGN, 2008] IGN, *Starcraft 10th anniversary interview*. Disponible en: http://www.ign.com/articles/2008/03/31/starcraft-10th-anniversary-interview?page=3 [2017, 13/06].

[Intelligent Systems, 1990] Anónimo, Wiki de Fire Emblem: Shadow Dragon and the Blade of Light. Disponible en: http://fireemblem.wikia.com/wiki/Fire_Emblem:_Shadow_Dragon_and_the_Blade_of_Light [2017, 13/06].

[JNI-BWAPI, 2015] *JNI-BWAPI*, repositorio oficial. Disponible en: https://github.com/JNI-BWAPI/JNI-BWAPI [2017, 19/05].

[Konami, 1999] Konami, Página oficial del juego de cartas *Yu-Gi-Oh!* Disponible en http://www.yugioh-card.com/es/ [2017, 13/06].

[Kosmos, 1995] Kosmos, página oficial. Disponible en: https://www.kosmos.de/spielware/spiele/catan/ [2017, 13/06].

[McCulloch y Pitts, 1943] McCulloch W. y Pitts W., *A Logical Calculus Immanent in Nervous Activity*, 1943.

[Mealy, 1955] G.H. Mealy, *A Method for Synthesizing Sequential Circuits*, 1955.

[MicroProse, 1991] Anónimo, Wiki de Civilization, página de Sid Meier's Civilization. Disponible en: http://civilization.wikia.com/wiki/Sid_Meier%27s_Civilization [2017, 13/06].

[Mike Robbins, 2012] Mike Robbins, Neural Networks in Supreme Commander 2. Disponible en: http://twvideo01.ubm-us.net/o1/vault/gdc2012/slides/Summit_AI/Robbins_Michael_Off%20the%20Beaten.pdf [2017, 21/05].

[Millington y Funge, 2009] Millington, I. & Funge, J. 2009, *Artificial Intelligence for games,* Segunda edición, Morgan Kaufmann.

[Moore, 1956] E.F. Moore, *Gedanken-experiments on Sequential Machines***,** 1956.

[OpenAge, 2017] *openAge*, página oficial.Disponible en: http://openage.sft.mx/ [2017, 30/05].

[OpenRA, 2017] *openRA*, página oficial. Disponible en: http://www.openra.net/ [2017, 30/05].

[Pirovano, 2012] Pirovano, M. 2012, *The use of Fuzzy Logic for Artificial Intelligence in Games*, Department of Computer Science, University of Milano, Milano, Italy.

[PLG, 2008] Planning and Learning Group, *Transparencias de la asignatura Automated Planning*, 2008.

[PLG, 2016] Planning and Learning Group, *Transparencias de la asignatura "Inteligencia Artificial en la Industria del Entretenimiento"*, Curso 2016-2017.

[Relic Entertainment, 2017] Relic Entertainment, página official de *Dawn of War*. Disponible en: https://www.dawnofwar.com/ [2017, 13/06].

[Rodríguez López, 2017] Rodríguez López, D.A. *Código del Agente*. Disponible en: https://github.com/Patataman/Goliat.

[SCAI, 2016] StarCraft AI. *Why not StarCraft 2*. Disponible en: http://www.starcraftai.com/wiki/Why_not_StarCraft_2

[Schwaber, 2004] Schwaber, Ken. *SCRUM Development Process*, 2004. Disponible en: http://www.jeffsutherland.org/oopsla/schwapub.pdf

[SSCAIT, 2017] *Student StarCraft AI Tournament*, página oficial. Disponible en: http://sscaitournament.com/ [2017, 16/05].

[StarCraft AI, 2015] StarCraft AI. Disponible en: http://www.starcraftai.com/wiki/Main_Page [2017, 14/06].

[Stuart y Russell, 2004] Stuart J. Russell, Peter Norvig, *Inteligencia Artificial - Un enfoque moderno,* Segunda edn, Pearson.

[Synnaeve y Bessière, 2011a] G. Synnaeve & P. Bessière 2011, *A Bayesian model for opening prediction in RTS games with application to StarCraft*, 2011 IEEE Conference on Computational Intelligence and Games (CIG'11), pp. 281.

[Synnaeve y Bessière, 2011b] G. Synnaeve & P. Bessière 2011, *A Bayesian model for RTS units control applied to StarCraft*, *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pp. 190.

[The All Seeing AI] Anónimo, TVTropes wiki. Disponible en: http://tvtropes.org/pmwiki/pmwiki.php/Main/TheAllSeeingAI [2017, 14/06]

[The Creative Assembly, 2000] The Creative Assembly, página official de *Total War*. Disponible en: https://www.totalwar.com/#games [2017, 13/06].

[TSP, 1800] W.R. Hamilton, *Traveling Salesman Problem* (TSP), Disponible en: https://en.wikipedia.org/wiki/Travelling_salesman_problem

[Unbehauen, 2009] Heinz D. Unbehauen, *Control Systems, Robotics and AutomatioN – Volume XVII: Fuzzy and Intelligent Control Systems*, Primera edición, 2009.

[Wargus, 2011] Página oficial de *Wargus*. Disponible en: http://wargus.sourceforge.net/index.shtml [2017, 30/05].

[Weber et al., 2010] Ben G. Weber, Mateas, M. & Jhala, A. 2010, *Applying Goal-Driven Autonomy to StarCraft*, Artificial Intelligence and Interactive Digital Entertainment (AIIDE).

[Wildfire Games, 2017] Wildfire Games, página official de *0 A.D.* Disponible en: https://play0ad.com/ [2017, 30/05].

[Wired, 2014] Golson, J., *In Forza Horizon 2, computers finally drive as crazy as humans*. Disponible en: https://www.wired.com/2014/09/forza-horizon-2-drivatars/ [2017, 13/06].

[Wizards of the Coast, 1993] Wizards of the Coast, página official de *Magic: The Gathering*. Disponible en: http://magic.wizards.com/es [2017, 13/06].